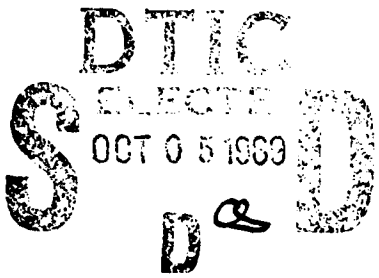
(2)

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

COMPRESSION OF
BITMAPPED GRAPHIC DATA

by

Jane Lee Kretzmann

June 1989

Thesis Advisor Uno R. Kodres

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (If applicable) 52	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 94943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 94943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
COMPRESSION OF BITMAPPED GRAPHIC DATA

12. PERSONAL AUTHOR(S)
Kretzmann, Jane L.

13a. TYPE OF REPORT	13b TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM _____ TO _____	June 1989	90

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense of the U.S. Government.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
data compression, compression, grahics, bitmapped, run-length encoding, Huffman codes, lossless, lossy, relative encoding, statistical encoding

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This paper explores the general topic of data compression, with emphasis on application of the techniques to graphic bitmapped data. Run-length encoding, statistical encoding (including Huffman codes), and relative encoding are examined and evaluated. A compression application of the Huffman coding of a run-length encoded file is designed and partially implemented in Chapter VII. A listing of the computer program which performs the compressio is included as an appendix. Possibilities for further study are suggested.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL Uno R. Kodres	22b. TELEPHONE (Include Area Code) (408) 646-2197	22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

Approved for public release; distribution is unlimited.

Compression of
Bitmapped Graphics Data

by

Jane Lee Kretzmann
B.A., The Colorado College, 1966

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
1989

Author: Jane L. Kretzmann
Jane L. Kretzmann

Approved by:
Uno R. Kodres, Thesis Advisor

Douglas G. Williams, Second Reader

Robert B. McGhee, Chairman
Department of Computer Science

K. T. Marshall
Kneale T. Marshall
Dean of Information and Policy Science

ABSTRACT

This paper explores the general topic of data compression, with emphasis on application of the techniques to graphic bitmapped data. Run-length encoding, statistical encoding (including Huffman codes), and relative encoding are examined and evaluated. A compression application of the Huffman coding of a run-length encoded file is designed and partially implemented in Chapter VII. A listing of the computer program which performs the compression is included as an appendix. Possibilities for further study are suggested.

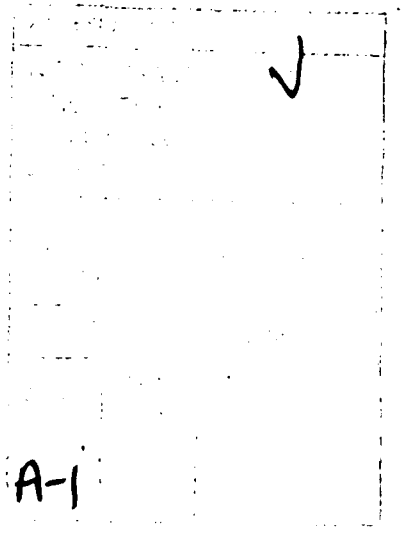


TABLE OF CONTENTS

ACKNOWLEDGEMENT

I wish to acknowledge my appreciation to the following persons for the support and encouragement which they provided:

Uno Kodres, my thesis advisor, who showed so much patience and provided direction down a path that had an end.

Douglas Williams and Roger Hilleary, my supervisors, whose continued encouragement and dedication to the importance of an individual's education have truly made it possible for me to succeed in this endeavor.

Michael Mayer, reader and friend, whose excellent critique, innovative ideas, and exceptional technical understanding have definitely enhanced this work.

Daniel Davis, advisor and friend, whose creativity and wisdom have sparked my own, and who consistently pointed me in the right direction.

Dave, Laurie, Tanner, Mike, and Cheryl, my family, who though mentioned last, are of the greatest importance in this endeavor and in my life. Without their continued support and understanding through too many months of separation, this thesis could not have been attempted.

I. INTRODUCTION

Computer graphics images have large data storage requirements. For example, the video memory bitmap[1] for a computer monitor with a resolution of 1024 by 1024 pixels requires one megabyte of memory to store an eight-bit grey-scale image. A 32-bit color image consumes four times the storage resource. The time taken to send such an image through a communication channel is significant. Therefore, methods which can reduce the size of a graphics file are of practical interest.

A. BACKGROUND

In the recent past, it was expected that graphics output and programs created on one computer would be used exclusively within the realm of that environment. Now, with the increasing popularity of computer networks, the capability exists for sharing resources among different computer systems. Indeed, Tanenbaum suggests that one goal of networking is "to make all programs, data, and other resources available to anyone on the network without regard to the physical location of the resource and the user." [Tan81: p.3] No longer is the computer user limited to software, hardware, and peripherals designed only for his particular machine.

The tremendous growth in the field of database management has meant an increase in large-scale information transfer by remote computing and the development of massive information storage and retrieval systems. Any method which reduces the size of the data for such systems implies a savings in the cost of data storage and transmission time. Thus, data compression techniques have gained popularity as a realistic means to accomplish these savings. [Hel87: p.1]

Likewise, graphics applications have become more sophisticated and gained popularity in networking environments. These applications consume substantial computer resources such as memory and processor time. It is often desirable to use

---

[1] Refer to Appendix A, Glossary, for a brief definition of unfamiliar terms.

mainframe resources to develop and perhaps store graphics, but to be able to view, save, and manipulate the graphic image on a personal microcomputer or workstation.

Because graphics files are often very large, the process of sending them from a host computer to a microcomputer via low bandwidth channels (such as phone lines) can be slow. If a user is interested in rapidly viewing many files (a graphics database, for instance) the time required to transmit the files may seem unreasonable. Current research in methods of compression are proving beneficial in significantly reducing the size, and therefore, the time to transmit an image.

B. OBJECTIVE

The goal of this thesis is to examine and evaluate several methods of graphic data compression which are used in the field of computer science. In addition, we will look at these methods in relation to transmitting graphic images from the Naval Postgraduate School's IBM 3033 mainframe computer to microcomputers in order to determine a reasonable method of reducing the image transmission time.

C. SCOPE OF THE THESIS

Graphics may be stored as vector images or as bitmaps. This thesis will only address graphic images stored in bitmapped format. Research in various methods of reducing the size of the transmitted image file fall into these categories: (a) compression methods of passing all the information, but taking fewer bits to do so ("lossless" compression) or (b) methods of reducing the amount of information passed through the network and reconstructing an incomplete image on the receiving computer ("lossy" compression). This thesis will investigate only the first category with emphasis primarily on techniques which can be applied specifically to the compression of binary (black / white) images.

D. OVERVIEW

Chapter II introduces several compression techniques in use in the field of computing. In subsequent chapters, these are discussed in depth, showing examples

of current use of these compression methods. Emphasis is on techniques used to compress graphical data.

Chapter III discusses the run-length encoding technique. Chapter IV explores statistical coding with particular emphasis on Huffman codes. Chapter V concerns other techniques, especially that of relative encoding.

Chapter VI is an the evaluation of the techniques discussed in the previous chapters. The main focus is on how these techniques can be applied to binary images.

Chapter VII describes the specific application of one technique of transmitting binary images in a network environment. The question is, "Can the compression techniques studied reduce the time to transmit digital images from the IBM 3033 mainframe to the IBM microcomputer by significantly compressing the file to be sent?"

Chapter VIII offers a summary of research findings and conclusions drawn from applying these techniques to the issue of binary image transmission.

A set of appendices is included to aid the reader in understanding details of the thesis. Appendix A is a glossary of terms which may not be familiar to the reader. Appendix B displays, in reduced format, the graphs which are included in the sample set used in the compression implementation in Chapter VII. Appendix C contains a listing of the programs written to perform the Huffman coding and to analyze the compression results. And Appendix D shows an example of an RLE (run-length encoded) file and the Huffman coding of the file.

II. DATA COMPRESSION METHODS

Davisson and Gray define data compression as "the science and/or art of massaging data from a given information source in such a way as to obtain a simplified or compressed version of the source data with at most some tolerable loss of fidelity." Areas where data compression has been used include systems for communications, speech and image processing, pattern recognition, information retrieval, storage, and cryptography. But the theory and practice of data compression began as early as 1898, when W. F. Sheppard studied the "rounding off" of real numbers to a fixed number of decimal places. [Dav76: p.1]

Because data compression, the substitution of data by a more compact representation, implies savings of resources (transmission time, storage media, computer memory, and money), it will always be a topic worthy of study. In the middle of this century, Shannon [Sha48], Fano [Fan49], and Huffman [Huf52] were researching improved methods of data compression.

In 1977, the National Bureau of Standards published a report on data compression to assist Federal Agencies in developing economical data element standards [Aro77: p.1]. In 1988, a paper published in ACM Computing Surveys assessed a variety of data compression methods spanning almost 40 years of research [Lel88]. Many of the techniques covered in both papers were based on the earlier work of Shannon, Fano, and Huffman, a tribute to the continued importance of these methods of data compression.

A. EXPLANATION OF BITMAPS

Basically there are two ways to represent a graphic in computer memory: as a vectorized image or as a bitmapped image. The first method stores the graphic information as sets of coordinates of the lines, or vectors, which define the image. The second method stores a bitmap of the image in memory. This thesis is limited to the domain of bitmapped graphic images.

A **bitmap** is a virtual representation of a specific screen image of the target monitor. For instance, an Enhanced Graphics Adapter (EGA) monitor which has a resolution of 640 pixels in the horizontal direction and 350 pixels vertically, contains a total of 224,000 pixels. If the monitor is monochrome, then each pixel has only two states, on or off, and may be represented by one bit in memory. The bitmap (bitplane, or monitor mapping) occupies 28 kilobytes of computer memory. The mapping of a color monitor is different.

Each pixel of a color monitor is composed of three colors: red, green, and blue. The intensity of each color varies to define different hues, while the combination of the intensities of all three colors designates the shades of pixel color. For instance, an IRIS-4D GT monitor has a resolution of 1280 by 1024 pixels. Each color may be set to 256 different intensities, requiring eight bits per color or 24 bits per pixel. Thus there are 16 million different colors available on this system.

How color graphics is implemented varies greatly from system to system. Even grey-scale graphics may take from eight to 24 bits to represent one pixel. Values range from black to white; black is represented by three values of the lowest intensity (0,0,0), white by three values of the highest intensity (256,256,256), and grey shades in between by equal values of mid-range intensities (100,100,100). [SGI: p.4-3]

In the computer graphics monitor industry, the number of horizontal and vertical addressable pixels on the CRT is called **resolution**. Resolution depends on the pitch and size of the phosphor dots on the CRT screen, and the brightness and purity of color that can be displayed. The size, spacing and number of dots is a function of the tube, but brightness and color quality depend on the monitor's electronics.

The following quotation shows the range of differences in monitors on the market today and indicates the varied hardware and software that must exist to support such diverse configurations.

> The 19-in. CRT is the de facto standard display size for engineering workstations. The large viewing area offers reduced eye strain. A monitor with a tube this size is considered to be ultra-high resolution if it has more than 1,280 horizontal addressable pixels. An example is 1,600 horizontal by 1,280 vertical. But if it has 1,000 horizontal pixels or above, it is considered to be a high resolution. A monitor with a 1,024 by 800 resolution is an example of this level.

The most popular monitors for PCs used in CAD/CAM and graphics application have screens measuring 13 or 14 inches diagonally. A monitor in this size range with 500 to 1,000 horizontal pixels is considered to be a high resolution. Sony, for example, offers a 900 by 560 monitor while NEC offers an 800 by 560 product.

...Among the state-of-the-art monitors now on the market are the Sun-4 workstation with 1,152 by 900 resolution and IBM's PS/2 system monitor with 1,024 by 768 resolution....

...Sony, received a custom contract from the FAA for a 37-in. graphics monitors to be used in upgrading the U.S. air traffic control system. ...The custom-made monitors with 2,000 by 2,000 resolution reportedly sold for $50,000 each. [Wil88: p.10]

Figure 2.1 shows typical resolutions for some of the popular graphics adapters on the market.

```
Low Resolution (LR):  128 x 128 to 510 x 510
Text only                MDA
320 x 240                CGA, MCGA

Medium Resolution (MR):  512 x 512 to 800 x 600
640 x 350                EGA
512 x 512
852 x 350                Super EGA
720 x 348                Hercules
640 x 480                VGA and PGA
800 x 600                Super VGA

High Resolution (HR):  801 x 601 to 1200 x 1023
1024 x 768               8514/A, Extended VGA
1280 x 800               Wyse and other DTP

Very High Resolution (VHR):  1201 x 1024 to 2048 x 2048
1280 x 1024              IBM's next controller
1600 x 1024
1680 x 1280
1200 x 1800              MCA (IBM's future controller)

Ultra High Resolution (UHR):  2049 x 2049 and above
3072 x 2048              UHR DTP systems
4096 x 4096              Vector displays
```

Figure 2.1. Resolution Segments [Ped89: p.8].

B. LOSSLESS AND LOSSY COMPRESSION

This thesis concentrates on compression techniques referred to as "lossless." Using these methods of compression, a file is compressed (encoded), transmitted, and decompressed (decoded) to produce a file identical to the original. The methods which will be discussed include run-length encoding, statistical encoding, and relative encoding.

Contrast this to methods of "lossy" compression where a graphics file is encoded such that an incomplete image is re-created in the decoding phase. Examples include fractal image compression, color compression, and spatial compression.

In fractal image compression, the shapes of natural objects in the original graphic image, such as trees, clouds, and fire, are numerically encoded using fractal geometry. These are then re-created as fractal images on the target machine. The technique is "lossy" because, although the decoded images closely resemble the original shapes, they are representations. [Win88: p.24] [Pet87] [Bar88]

Another method of compressing color images is to reduce the number of bits per pixel normally available to a system. This technique limits the maximum number of colors from, say, 256 (16-bit pixels) to 16 colors (four-bit pixels), but is effective where color does not play an integral part in the identification of the image. A satellite image is a good candidate for color compression, whereas a graphic design of molecules which are distinguished by their color may not be. [Mur88b]

In some instances color definition may be important, but a fine resolution may not be. Spatial compression takes advantage of this property. Consider a bitmap which represents a screen resolution of 512 by 512 pixels, or 256 kilobytes for an eight-bit grey-scale image. Each four by four block of pixels is replaced with one pixel containing a weighted average of the intensity values of the 16 pixels in the original block. Using this technique the bitmap can be reduced to 128 by 128 pixels, or 16 kilobytes, although the decoded image may appear fairly ragged. If the spatially-compressed image is recognizable, then the achieved compression ratio is 16:1. This type of lossy compression is useful in browsing a large database of graphic images. When the desired image is located, it may be transmitted by a lossless method and completely reconstructed on the target monitor. [Mur88a]

III. RUN-LENGTH ENCODING

Run-length encoding compresses data by taking advantage of a **run**, or series of the same value occurring consecutively. A run may be any of the following: a repeating single-bit value in a black and white bitmapped image file, a repeating character in a text file, or a repeating pixel value in a color graphics file. A run may even be a larger pattern which repeats itself a number of times. For instance, consider that a repeating number is actually a repeated pattern of eight bits; notice the repeated pattern in a bitmap which uses patterns of black and white bits to simulate a shade of grey; and realize that blocks of pixels which are repeated may also be considered a run of patterns.

Regardless of the type of data (ASCII characters, binary data, etc.), run-length encoding is guaranteed to reduce the physical size of the file if the length of each run is greater than the number of bytes or bits substituted for the original sequence in the compressed file.

A. TERMINOLOGY

It is appropriate at this point to define the terminology which is used in this section. As will be explained in greater detail, a run is encoded by the following elements:

- **compression indicator character:** any seldom-used predefined character or bit pattern which indicates that compressed data follows.
- **run value:** a bit, a pixel, a character, or a pattern which is repeated.
- **run length:** the number of times the run value is repeated.

An important concept which will be used repeatedly is that of a **minimum run length**. This is the minimum number of consecutive values that must be in a run for run-length encoding to be beneficial. In other words, it is the break even point between an increase in file size caused by the three elements just mentioned, and a savings in file size made possible by compression.

B. COMPRESSING DATA FILES USING NULL SUPPRESSION

Null or blank suppression is a simple type of run-length encoding, and represents one of the earliest uses of data compression. It is particularly useful on files which contain fixed-length records, such as language source programs and database files. Null suppression reduces repeated occurrences of the null or blank character to two bytes. One byte contains a compression indicator character, usually an unprintable character or seldom-used character chosen from the character set (ASCII, EBCDIC, etc.) used by the file. If the compression indicator character does appear in the original data file, it can be made unambiguous by doubling its appearance in the compressed file. The second byte contains the number of null characters in the run. The upper limit of 255 (the maximum value which one byte can represent) is adequate for a text data file.

Figure 3.1 illustrates a simple file compressed by blank suppression. Notice that

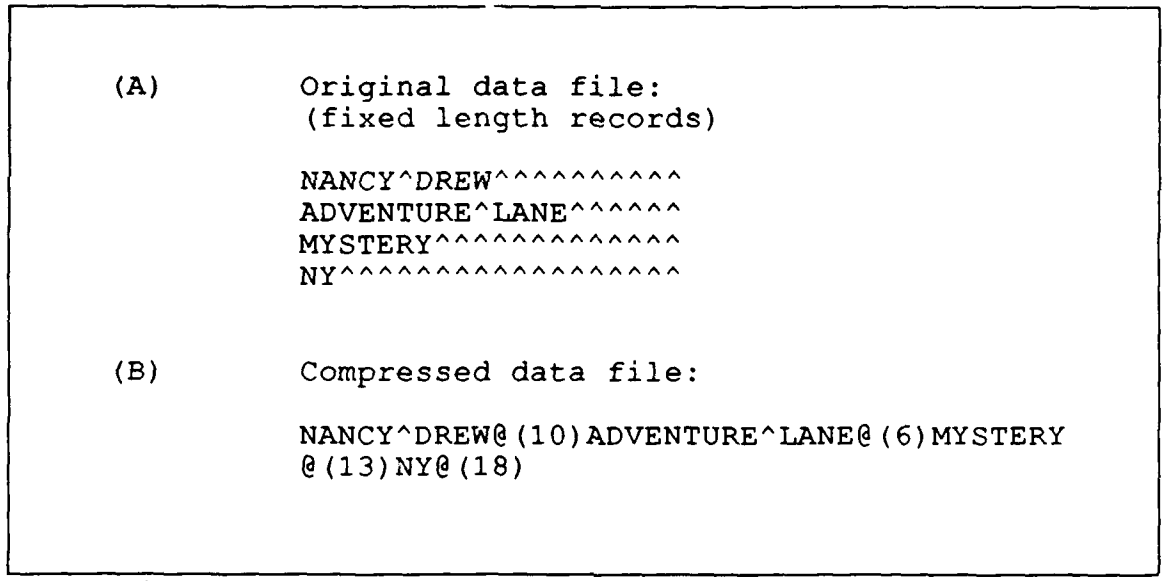


Figure 3.1. Compression by Blank Suppression.

the numbers in parentheses are decimal representations of the bit configuration of the run length and only occupy one byte. The "^" character is used to represent the blank character.

The original file requires 80 bytes, whereas the compressed file contains only 41 bytes. This is a compression ratio of nearly 2:1, or 49%. It is perhaps more significant, however, to only consider the compression on the blank characters of the file; in this case, 50 blank characters were replaced by ten bytes of encoded data, producing a compression ratio of 5:1, or 80%. It is clear from this example that no advantage is gained from compressing runs of fewer than three blanks. Therefore, the minimum run length for null or blank suppression is three characters.

Example. NARC.EXE is a public domain shareware product for microcomputers written by Gary Conway. It is a de-archiving facility for storing files in a compressed format. Several storage methods are available to allow the user to "pack," "squeeze," "crunch," or "squash" a file. "Packing" is an implementation of blank suppression. Conway says that

> [Packing] is the simplest of the storage methods. Suppose that you have a line of text and at the end of the line, you have 40 spaces. These 40 spaces are compressed into 3 bytes in the ARC file. The first byte is the actual character to be expanded (in our case a space). The second byte is a special "flag" byte that indicates that we need to expand these bytes. The third byte is the count byte (in our case it would be 40). So you can see that any time the ARC'er finds repeated bytes like this, it can compress them into 3 bytes. [Con87: p.9]

Notice that the NARC method requires three bytes to compress a run, compared to the two bytes described previously. While the NARC technique has the disadvantage of taking more space and not gaining the maximum compaction available through null suppression, it has the advantage of being more general, as does the following method.

C. COMPRESSING ASCII DATA FILES

To use run-length encoding on an ASCII data file requires not two, but three, bytes of information: one byte for the compression indicator character, another byte for the run length, and a third byte for the value of the character being repeated. For this type of application, the minimum run length is four characters and the maximum run length is 255 bytes. Also, compression might be feasible if a file of alphanumerical data contained patterns of characters, such as a repeating number. However it appears that run-length encoding is not a good candidate for compressing an English text file other than through null or blank suppression.

D. COMPRESSING EIGHT-BIT GRAPHICS FILES

Run-length encoding is particularly beneficial when the information represented is a bitmapped graphics file. Let us first look at a type of graphics file which contains eight-bit codes to determine the feasibility of run-length encoding. This is the grey-scale or eight-bit color graphics data file; each pixel in the bitmap is represented by eight bits, or one byte. Again, the minimum run length is four pixels, but finding four or more consecutive pixels of the same shade or color is highly probable.

Refer to Figure 3.2 for an example of this implementation. In this figure, the

```
(A)  Original bitmap:

                BBBBBBBBBBBBBBBBBBB
                wwwwwRwwwwwwwwGwwwwwww
                wwwwRRRwwwwwGGGwwwwww
                wwwRRRRRwwwGGGGGwwwww
                wwwwRRRwwwGGGGGGGwwww
                wwwwwRwwwwwwwwBwwwwwww
                BBBBBBBBBBBBBBBBBBB

(B)  Compressed data:

     @(20)B@(5)wR@(7)wG@(10)wRRR@(5)wGGG@8w@(5)Rwww
     @(5)G@(8)wRRRwww@(7)G@(8)wR@(7)wB@(6)w@(20)B
```

Figure 3.2. Compression of Graphics Data.

small bitmap (A) shows a red diamond and a green tree on a background of white with black borders. The original bitmap takes 140 bytes. The compressed file (B) uses 61 bytes. This is greater than 56% compression.

Again, the numbers in parentheses are decimal representations of the run length and only occupy one byte. The run length contains values from four to 255 pixels because a minimum run length of four pixels indicates that no compression will occur on runs of length zero through three. Conceivably this byte could be used to contain run lengths from four to 259 pixels, thereby increasing the maximum run length coded in one byte. This technique of increasing the maximum capacity of a byte in this

manner may be applied to other examples as well. If a run is greater than the maximum permitted, it may be expressed as several runs of the same value. For instance, a run of 400 red pixels may be compressed into the following six bytes: @(259)R@(141)R.

Example. GEM™, a software product of Digital Research Inc., stands for Graphics Environment Manager. It offers run-length encoding as one option for compressing a bitmapped file where each pixel occupies a variable number of bits, say eight. In this particular situation, encoding is implemented as follows.

A "run-length packet" is used to represent a run of less than 128 pixels. This packet consists of two bytes of information: the length of the run and the eight-bit value of the pixel. Since a bitmap is logically one long string of information, a single run of pixels may be longer than a line on a monitor.

For a run greater than or equal to 128 pixels, an "extended run-length packet" is used. This is a three-byte packet containing the following information: an opcode of value -1, an extended run byte containing a count of 128-pixel runs, and the pixel value. Figure 3.3 illustrates this concept.

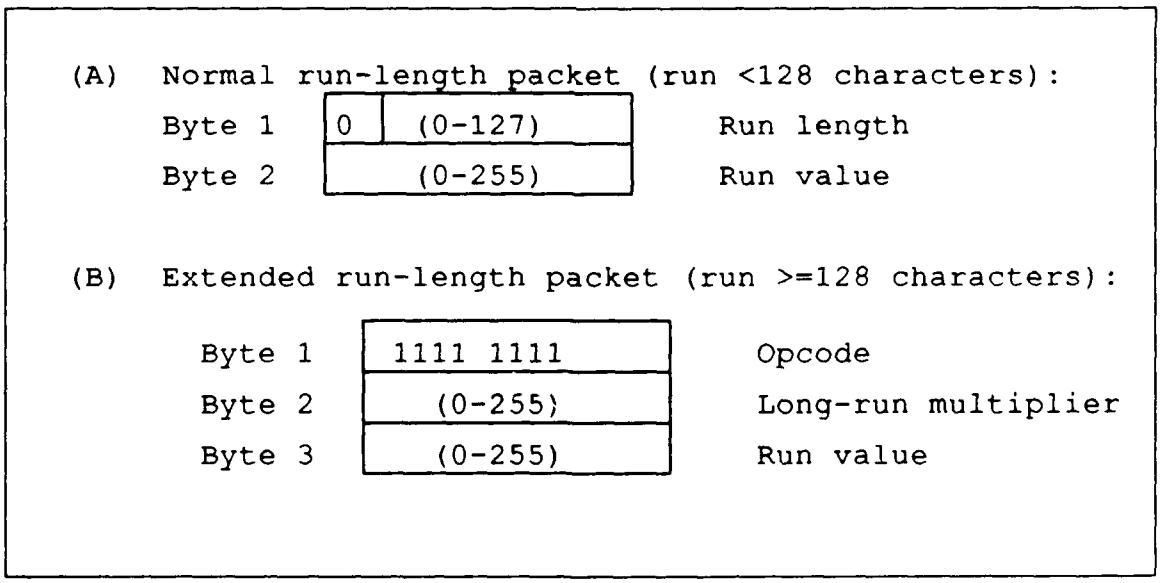


Figure 3.3. Run-length Packets in GEM™.

Figure 3.4 shows an example of a run consisting of 1000 pixels in length. An

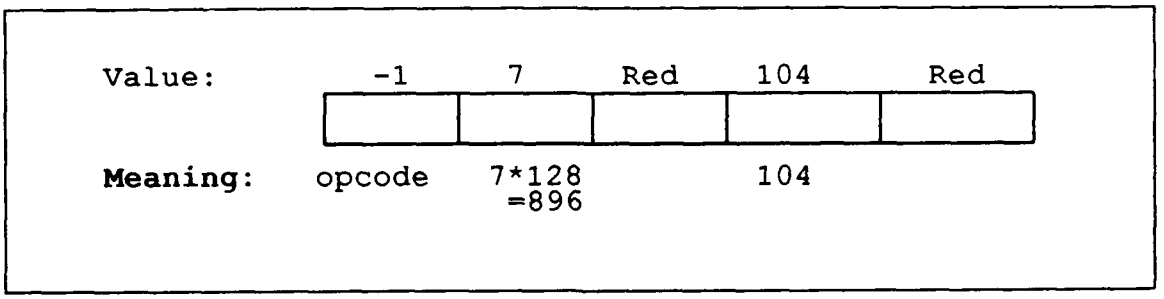


Figure 3.4. Example of Long Run in GEM™.

efficient way to encode this is to use an extended run of length seven (7 * 128 = 896 pixels) followed by a standard run of length 104. [DRI85: p.I-2]

For completeness, it should be mentioned that under this implementation the entire file is encoded. However the encoding method also includes options other than run-length encoding, such as pattern encoding which is indicated by an opcode value of -3.

E. COMPRESSING BINARY IMAGE FILES

Some interesting variations of run-length encoding exist for a binary image file. If graphics data contains only black and white values, then each pixel can be represented by a single bit. Compressing such a file using normal run-length encoding as previously described, is not the best method.

For instance, in our previous example, a pixel was represented by one byte of data; a minimum run length of four pixels or less provided reasonable compaction. With binary data, using the same implementation yields a large minimum run length. The three bytes required to compact one run could hold 24 pixels. Thus, in order to benefit from compaction, a run must be at least 25 pixels long.

Depending on the size of the bitmap and the degree of uniformity of the graph, such an implementation may render acceptable compression. If, for instance, a bitmap of 1024 by 1024 pixels contains mostly white space, or long, horizontal black lines, the long runs would qualify for compaction. But variations of run-length encoding offer greater efficiency for binary image data.

The first decision to make when considering compression of a file is "to compress or not to compress." Unless a file contains a high percentage of runs which exceed the minimum run length, the human and computer resources required may not be worth the effort.

If we can decrease the number of bytes required to compress a run, and hence reduce the minimum run length, then a file may more easily qualify for compression. There are several techniques for accomplishing this objective.

Method One. One technique is to encode the entire data file, regardless of the length of a run. This decision immediately eliminates the need for the compression indicator character, and reduces the minimum run length from 25 pixels to 17 pixels.

In addition to deleting the indicator byte, if we then combine the length of a run and its value into one byte, we have further reduced the minimum run length to nine pixels. With this encoding scheme one byte of compressed information would appear as follows:

bit one: value ("0" or "1")

bits 2-8: run length (0 - 127)

Runs longer than 127 would simply require two or more bytes to represent them.

Method Two. Alternatively, by using the entire eight bits of a byte of compressed data, we could represent a maximum run length of 255 pixels. Since the entire binary file is being encoded, why not simply transmit a series of run lengths? If we make certain assumptions about the compressed file, then this is possible. Let us assume that the first value transmitted is always zero ("0"). Let us also assume that for a run greater than 255 of a particular value, a "null" byte of run length zero for the opposite value will be interjected between bytes showing the longer run. While this method achieves better compression on runs of lengths 128 to 255 bits, the overhead incurred by the null byte for runs greater than 255 bits reduces the efficiency of the method.

Method Three. In order to make maximum use of each byte for compressing data, and yet assume transmission of alternating values, beginning with a "0" bit, we add another modification. To handle the problem with long runs, implement a "base 127" approach. Runs of length zero to 127 bits are represented by a single byte

containing that value. If a run exceeds 127 bits, then use values from 128 through 255 to represent a multiple of 128. Notice that for these values the leftmost bit of the compression byte is "1." This bit turned on signifies that two bytes, rather than one, contain the run-length value. This concept is similar to that used in GEM™ in the example above. This variation on run-length encoding is also the compression technique used by Michael Gunning in the GRAFPC program which will be examined in Chapter VII.

Figure 3.5 compares Methods One, Two, and Three, showing the different number of bytes required to compress a sample file. In this figure, the numbers in parentheses represent the run length contained in one compression byte. In Method One, the run length value is shown as the first bit of the compression byte. In Methods Two and Three, a null byte is transmitted first since the assumption is that a compressed bit stream begins with a zero value.

In conclusion, we can infer from the example that, as the length of the runs increases, so will the number of bytes required by Methods One and Two, but Method Three will never require more than two bytes to represent a run. However, all three methods give excellent compression.

F. ENCODING PATTERNS

Another important variation on normal run-length encoding considers a run of any pattern to be a candidate for compression. One possible method for encoding a pattern requires that four items of information be used for each run:

- A compression indicator character to indicate the start of a run
- The length of the run
- The length of the pattern
- The pattern

The pattern length must be included because it is variable.

This approach will work for an ASCII data file which is inspected byte by byte. However, it is a more complicated problem to encode a raw bitmapped data file, where a string of bits may appear in any conceivable configuration. For instance, consider a dithered bitmap which simulates grey-scale filled area with runs of the

Run Value	Run Length	Method One	Method Two	Method Three
"0"		(0)		(0)
"1"	50	(50)	1(50)	(50)
"0"	100	(100)	0(100)	(100)
"1"	200	(200)	1(127) 1(73)	(128) (72)
"0"	300	(255) (0)* (45)	0(127) 0(127) 0(46)	(256) (44)
"1"	400	(255) (0)* (145)	1(127) 1(127) 1(127) 1(19)	(384) (16)
"0"	500	(255) (0)* (245)	0(127) 0(127) 0(127) 0(119)	(384) (116)
"1"	600	(255) (0)* (255) (0)* (90)	1(127) 1(127) 1(127) 1(127) 1(92)	(512) (88)
Number of bytes to compress 2150 bits:		18	20	13
Compression:		93.3%	92.6%	95.2%

* Indicates a Null byte of the opposite value.

Figure 3.5. Compression of a Binary Image File Using Run-Length Encoding.

repeated pattern "on-on-off." In such a file, how can we embed, and expect to recognize, a unique indicator character?

One method is to define an arbitrary bit pattern to represent an indicator character, e.g., "11111111." In order to distinguish this indicator from the appearance of the same string in the raw data, the protocol of doubling an original occurrence is used. In other words, if "11111111" appears in the original bitmap, aligned on a byte boundary, then two identical bytes of "11111111" are transmitted. Next the file is processed byte by byte, searching for an indicator pattern. If not found, the byte is transmitted as raw data; if found, the next byte is checked to determine if this is an indicator or raw data. If this byte is the indicator, then the following data is treated as encoded data. A second indicator character indicates the end of compression. [May88]

IV. STATISTICAL CODING

Run-Length encoding generally requires that a fixed number of bits or bytes be used to represent a series, or run, in the file to be compressed. Now we consider a situation where variable-length codes may be used. The main idea inherent in statistical coding is that **if some symbols occur more frequently than others in a message, then we can take advantage of this fact by using a variable-length code such that frequent symbols will be replaced by shorter codes, while symbols which are used less frequently will be replaced by longer codes.** This concept is used in the Morse coding system, for instance. The most common letter in the alphabet, E, is represented by a single "dot," whereas a Q is transmitted as "dash-dash-dot-dash."

A. TERMINOLOGY

In this discussion of statistical coding, the following terminology will be used to identify the data to be compressed. The entire file to be compressed is referred to as the **message**. Once encoded for transmission, the message becomes the **encoded message**.

The information elements of a message are referred to as **symbols**. A symbol may be a character of the alphabet, it may be a bit representation of picture elements (pixels) in a graphics file, or it may even be a group of characters or bits. The set of symbols used in a message is referred to as S_1, S_2, S_3, ... S_n, where n is the cardinality of the set. Each symbol, S_i, has associated with it both a length, L_i, and a probability, P_i. The length is the number of bits used to encode the symbol, whereas the probability indicates how often the symbol is used in the message.

The **source alphabet** of a file is the set of all possible symbols which may be used in a message. Examples include the set of ASCII characters, the lower-case English alphabet, and the set of all colors that can be represented by eight bits.

Source symbols are a subset of the source alphabet; these are the symbols which are actually used in the message. One example would be all the colors present in a graphic image: 58, say, as opposed to the possible 256 colors available.

Code symbols are the variable-length strings assigned to encode the source symbols; these are generally listed in a **decoding table**. The elements, or **coding digits**, which are used to compose a code symbol are "0" and "1" in the binary number system, which has a **radix** of two. However systems of a radix other than two are used; for instance, the Morse codes use coding digits from the set {"dot", "dash", "pause"} which has a radix of three [Ham86: p.15].

Lastly, a **code** may be thought of as a mapping of source symbols onto code symbols. The decoding table shown in Figure 4.1. contains a sample code. The act of exchanging the set of source symbols which make up the message into the set of code symbols which compose the encoded message is referred to as **encoding**. A similar process, in reverse, will generally **decode** the encoded message back to the original message.

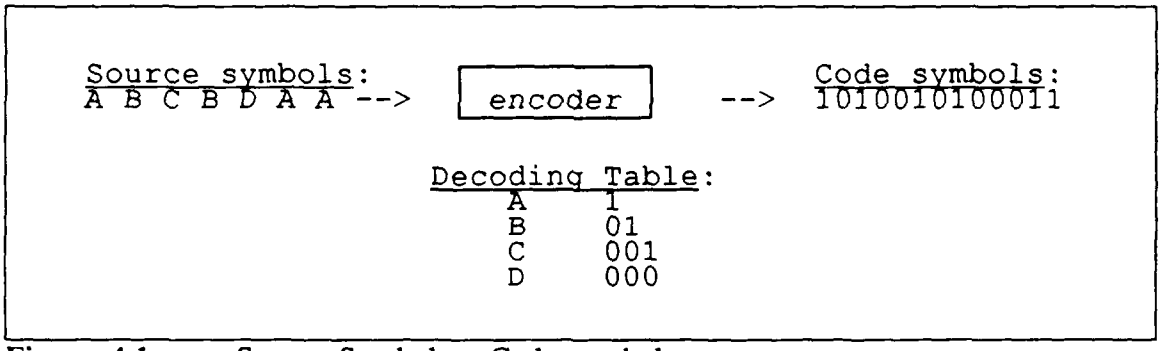


Figure 4.1. Source Symbol to Code symbol.

B. CODING AND INFORMATION THEORY

The presentation of some background in the coding theory which underlies statistical compression methods will provide us with a measure for evaluating the effectiveness of these methods in general, and Huffman's method in particular. If we view information theory, as Lelewer suggests, as the study of efficient coding and its consequences on the speed of transmission and probability of error, then we perceive

the primary objective of data compression as minimizing the amount of data to be transmitted. [Lel88: p.261]

1. Differentiation of Codes

The following terms are used to differentiate among codes. Also several properties exist which should be incorporated into the design of a code.

First, it is desirable to be able to distinguish one code symbol from another. In other words, we want to establish a one-to-one correspondence in mapping the set of source symbols onto the set of code symbols. Such a code is called **distinct**.

One problem in using variable-length codes is how to determine the end of one code and the beginning of another. A code is **uniquely decodable** if it is distinct and each code symbol embedded in an encoded message can be readily identified. A code in this category produces an encoded message which can have only one possible interpretation. In order to use variable-length codes, the code must have unique decodability. To use shorter codes for symbols with a high probability implies the use of variable-length codes.

But it is also desirable to know immediately when a complete symbol has been received, without having to examine other transmitted symbols before deciding what code symbol was sent. Morse codes use a "pause" of a predefined length as a code delimiter to meet this requirement. Another method is to insure that the code selected has the prefix property, which assures that **no encoded symbol of this code is a prefix of any other symbol**. If a code that is uniquely decodable has the prefix property, it is said to be **instantaneously decodable** and is referred to as a **prefix code** or an **instaneous code**. [Ham86: pp.52-56][Lel88: p.264]

Figure 4.2 illustrates the inherent hierarchical structure of the codes described. Figure 4.3 shows examples of codes which fall into the categories identified in Figure 4.2.

2. A Finite Automaton

One tool for determining instantaneous decodability is a finite automaton, which can also be represented as a decision tree, or a "decoding tree." This concept will become clearer through the use of an example.

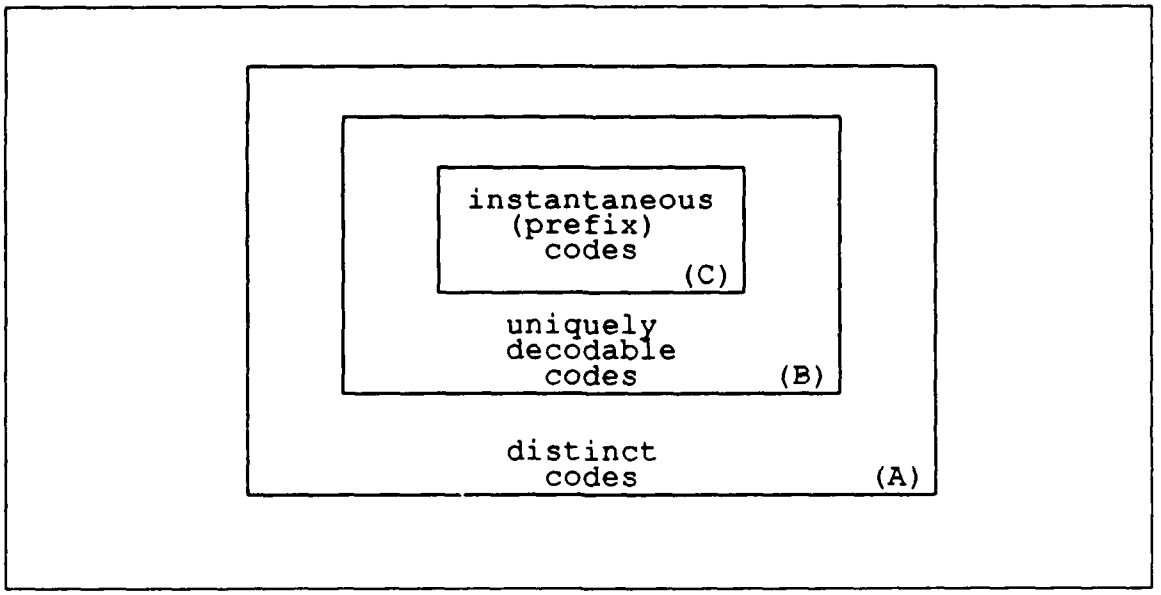


Figure 4.2. Hierarchy of Codes.

	(A)	(B)	(C)
	Distinct	Uniquely Decodable	Instantaneous
S_1 =	0	0	0
S_2 =	01	01	10
S_3 =	11	011	110
S_4 =	00	111	111

Figure 4.3. Examples of Codes of Different Categories.

Suppose that our message contains only four symbols, S_1, S_2, S_3 and S_4, and that each symbol is represented by some binary code, i.e., the coding digits are "0" and "1." Assume the following assignments are made.

$S_1 = 0$
$S_2 = 10$
$S_3 = 110$
$S_4 = 111$

Notice that the prefix property is preserved in these assignments since no code symbol is the prefix of another code symbol.

Next consider the finite automaton representation of this assignment in Figure 4.4 [Ham86: p.54]. Each node (or vertex) represents a state and each arc (or edge) a transition between the nodes it connects. Every arc is labeled with "0" or "1," depending on whether it is a left-handed or right-handed branch. It is arbitrary whether all branches in a particular direction are denoted by "0" or "1." The code for any particular symbol is obtained by following the path from the start state to the state where that symbol is defined, and by recording the labels of the arcs encountered along the way. If following the path of a code does not lead to a final, or "accepting" state, then the code does not preserve the prefix property.

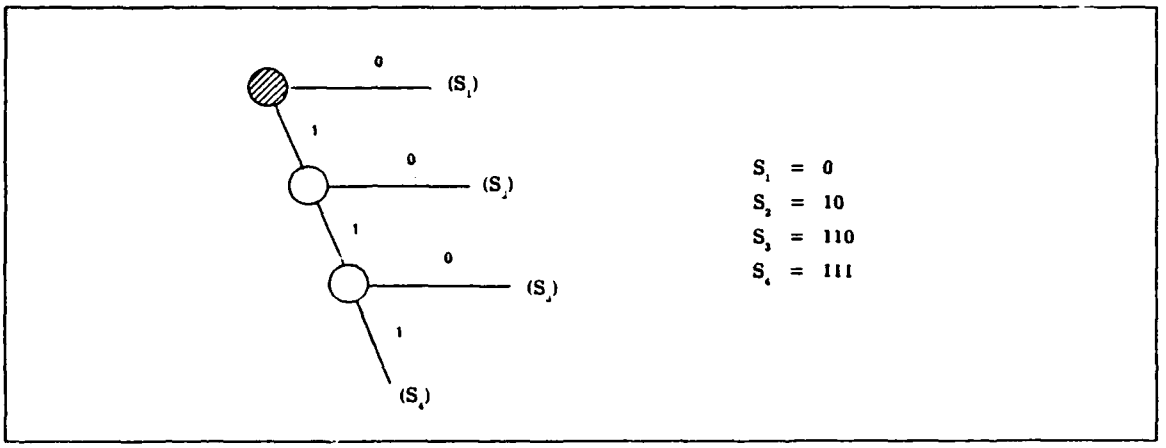


Figure 4.4. Finite Automaton.

Given the above explanation, one may wonder why the following finite automaton, or decision tree, would not work as well. This tree also preserves the prefix property. Both trees illustrate codes which are instaneous.

While it is true that both trees meet the prefix criterion, we must ask which symbol encoding will produce the better performance? The answer lies in knowledge of the statistical makeup of the message. We must answer the question "What is the probability of occurrence of each symbol in the message?" If the frequency of occurrence is evenly distributed, that is, all symbols occur with equal probability (like tossing a coin), then the tree in Figure 4.5 is better. But if S_1 occurs more frequently than S_3 or S_4, then perhaps we can capitalize on the fact that more of the message will

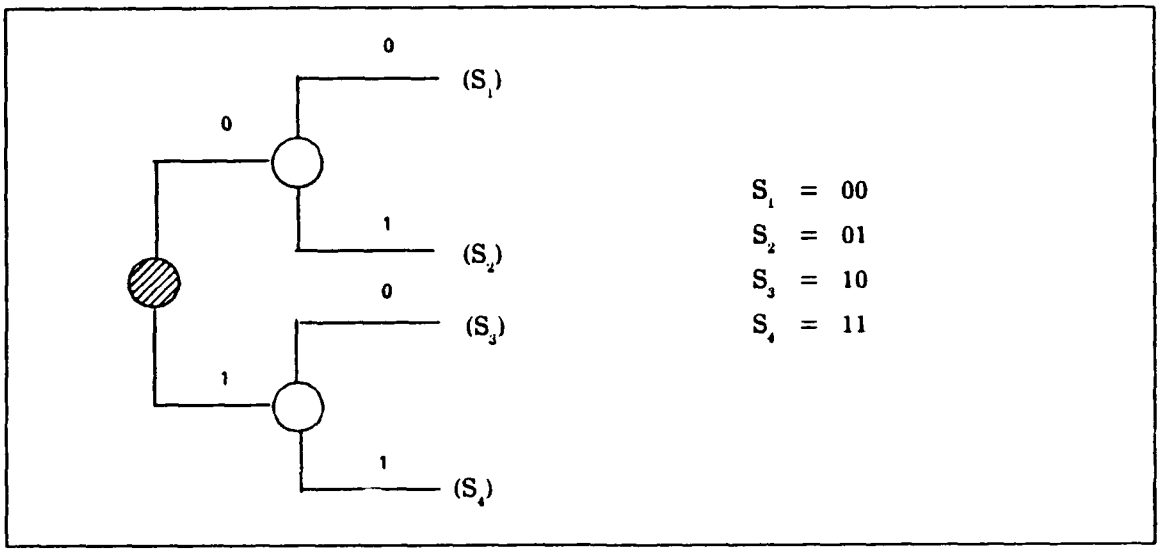


Figure 4.5. Binary Tree.

be represented by a one-bit code than by a three-bit code, thus producing a higher degree of compression and a shorter encoded file for transmission. Later examples of Huffman code implementation will demonstrate this fact.

3. Noiseless Coding Problem

Statistical compression techniques are generally approximations to the solution of the noiseless coding problem. Assuming a noiseless channel allows for a system in which the code symbols can be transmitted from one point to another without possibility of error [Lel88: p.262]. The goal is to construct a uniquely decipherable code which also minimizes the average length of the code symbols, where the average length is a function of the probability and length of each symbol. The formula is expressed as $L_{avg} = \sum P_i L_i$. Such a code is referred to as an **optimum code**. The ability to produce an optimum code is valuable because the shorter the length of the code symbols, on the average, the shorter the message, and therefore the shorter the time required to transmit the message, which is our ultimate goal in compressing data for transmission.

From the two decoding trees above, we will construct an example to use throughout this section. [Dav88] Given the four symbols from Figure 4.4, we assign arbitrary probabilities of occurrence to each symbol, and do likewise for the symbols from Figure 4.5. In Figure 4.6, situation (A) shows that the symbols occur with

unequal probability, but for situation (B) all symbols occur with equal probability. The average code-symbol lengths are calculated for each circumstance.

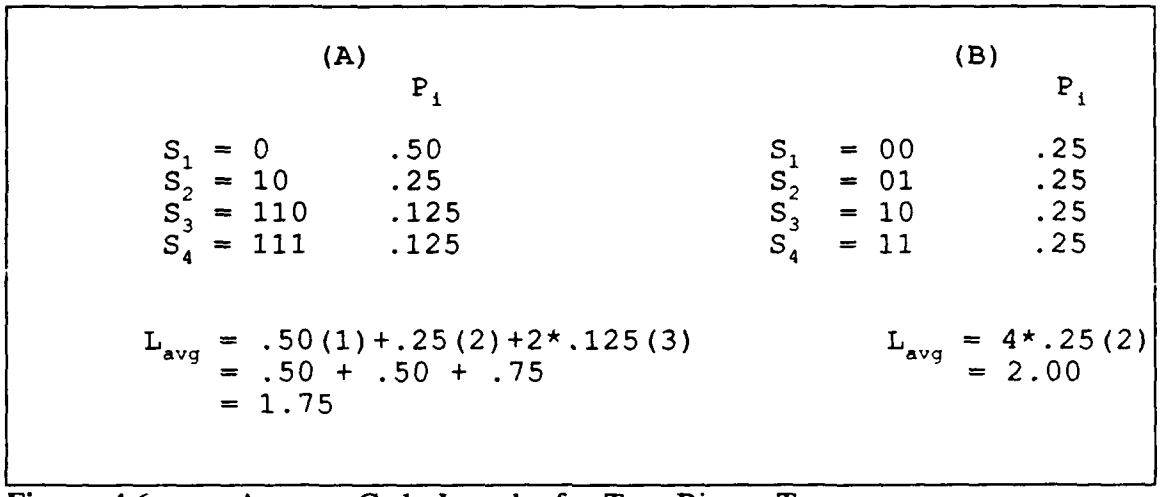


Figure 4.6. Average Code Lengths for Two Binary Trees.

We can see that the two codes have very different average code-symbol lengths with situation (A) being smaller than that in (B). But how do we know that the code in (A) is an optimal code?

4. Entropy

The degree of optimality of a code can be measured by the **entropy** of the message [Aro77: p.17]. The formula for calculating entropy is

$$H = -\sum P_i \log_2 P_i = \sum P_i \log_2 (1/P_i).$$

This value provides a lower bound on the average code length for an optimal code. In other words, a code may have an average code-symbol length very close to, but not less than the entropy. If the average length of the code symbols compares favorably to the entropy value, then the code is considered to be optimal.

Then what is entropy? Webster's Dictionary defines entropy as "a measure of the amount of information in a message that is based on the logarithm of the number of possible equivalent messages." Hamming describes entropy as "the average information of the [source] alphabet." [Ham86: p.108] He states, "The entropy function measures the average amount of uncertainty, surprise, or information that we get from the outcome of some situation, say the reception of a message..." [Ham86: p.114]

For instance, if the probability of a source symbol, say S_3, is equal to one, then the probabilities of the other symbols in the alphabet equal zero. Therefore, since from all possible symbols in the source alphabet it is certain that the next symbol we receive is S_3, then there is no surprise about the transmitted message, and hence no information; the entropy is $-\log_2 1$, or zero.

If, however, the source symbols have a probability distribution as in Figure 4.6 above, the transmitted message is uncertain and does contain information. Calculating the entropy of examples (A) and (B) we get values of 1.75 and 2.0, respectively. This is because in these examples the length of the symbol happens to equal the $\log_2$ of the probability of the symbol. The fact that the calculated values for L_{avg} and H are the same illustrates the high degree of optimality of these particular codes.

Consider a similar code in Figure 4.7 where this condition does not hold.

(C)

	P_i
$S_1 = 0$	.50
$S_2 = 10$	.30
$S_3 = 110$	.15
$S_4 = 111$	.05

$$L_{avg} = .50(1)+.30(2)+.15(3)+.05(3)$$
$$= .50 + .60 + .45 + .15$$
$$= 1.70$$

$$H = -(.50(-1)+.30(-1.74)+.15(-2.74)+.05(-4.3)$$
$$= .50 + .52 + .41 + .22$$
$$= 1.65$$

Figure 4.7. Example of an Optimal Code.

The entropy (H) for this example is calculated as 1.65 and the average code-symbol length (L_{avg}) is 1.70. This code is not as optimal as those in Figure 4.6.

C. HUFFMAN CODES

In 1952, Dr. David Huffman, published a paper entitled "A Method for the Construction of Minimum-Redundancy Codes." [Huf52] Building on earlier work by C. E. Shannon [Sha48] and R. M. Fano [Fan49], Huffman produced a method to derive an optimum code which results in the shortest average code length of all statistical encoding techniques [Hel87: p.97]. In his paper Huffman defines five basic restrictions which an optimum code must meet. They are quoted below. The reader should be aware that Huffman's terminology differs somewhat from that presented earlier. He uses the term "message" or "message code" for "source symbol."

(a) No two messages will consist of identical arrangements of coding digits.

(b) The message codes will be constructed in such a way that no additional indication is necessary to specify where a message code begins and ends once the starting point of a sequence of messages is known.

(c) L(1) <= L(2)<= ... L(N-1) = L(N).

(d) At least two and not more than D of the messages with code length L(N) have codes which are alike except for their final digits.

(e) Each possible sequence of L(N)-1 digits must be used either as a message code or must have one of its prefixes used as a message code.[2]

Restriction (b) is the requirement which demands "unique decodability." Restriction (c) assumes that messages are ordered with the probability decreasing and the length of the code for the message increasing. This restriction states that in order to have an optimum code, it is necessary that the length of the last symbol in the list (the one with the lowest probability of occurrence) equals the length of the next-to-last symbol. In restriction (d), "D" is the radix, that is, the number of coding digits available for encoding a symbol. For a binary coding system such as that used in the examples of this thesis, D always equals two.

---

[2] After constructing numerous Huffman trees, this author is of the opinion that restriction (e) as stated by Huffman is incomplete. The author believes it should be altered as follows:

(e) Each possible sequence of L(N)-1 digits must (1) be used as a message code, (2) have one of its prefixes used as a message code, or (3) must itself be the prefix of another message code.

Indeed, one and only one sequence of digits will be the prefix of all of the last D message codes in the list, as established by restriction (d).

1. Description of Technique

Having stated these restrictions, let us now look at the creation of a Huffman coding scheme. The process consists of the following steps:

- Define the frequency distribution of source symbols.
- Construct a Huffman code for the source symbols.
- Encode the message.
- Transmit the message.
- Decode the message.

Step One. The first step is to define a frequency distribution; this entails assigning a probability of occurrence to each symbol used in the message. One method of doing this is to scan the input file, tabulating data for each symbol and building a table of probabilities in the process. This table will be used in Step Two, refined, and transmitted with the encoded file to be used in the decoding process.

Another method of obtaining frequency information is to use a predefined table which was developed for a source alphabet similar to that used in the message. For instance, tables already exist which define usual probabilities of occurrence for each letter in the English alphabet. This table would suffice for large text files. A different table would be required if the symbols were words in a programming language.

Step Two. Constructing a Huffman code from source symbols is the next step. Our objective is to build a tree in which the nodes represent probabilities. The leaf nodes will be labelled with the original probabilities associated with each source symbol in the message, internal nodes will consist of derived probabilities as described in the following steps, and the root node will have the unity probability of 1.00. The arcs of the tree are each labeled with one coding digit, either "0" or "1." Theoretically the labelling is arbitrary, but for our example, let us label all branches toward the top as "0" and branches toward the bottom as "1." Reading the path from the root node to a leaf node will provide the Huffman code for each symbol. Adhering to the restrictions imposed above, we next describe the technique for building a Huffman code:

- Arrange the symbols by decreasing probability, with the symbols least likely to appear in the message at the bottom of the list. Restriction (c) above states that the two least frequent source symbols have the same encoded length. We begin with the probabilities of these two symbols.
- Form a new node by summing the probabilities of the two nodes at the bottom of the list. The two nodes combined will always be the two which have the lower probabilities on the current list.
- Make a new list of ordered probabilities with the new node inserted into a proper position in this list. Note that if one or more nodes already exist which have the same probability as the new node, it does not matter whether the new node is placed before or after the existing node(s). Huffman states that "it is possible to rearrange codes in any manner among equally likely messages without affecting the average code length." [Huf52: p.1100]
- Repeatedly perform the two previous steps until the list contains only two nodes. The sum of these two probabilities will equal one; this is the root node and we have built the desired encoding tree. Refer to Figure 4.8. It can be seen that for a source alphabet containing n symbols, the resulting tree will contain n levels, including the root node.

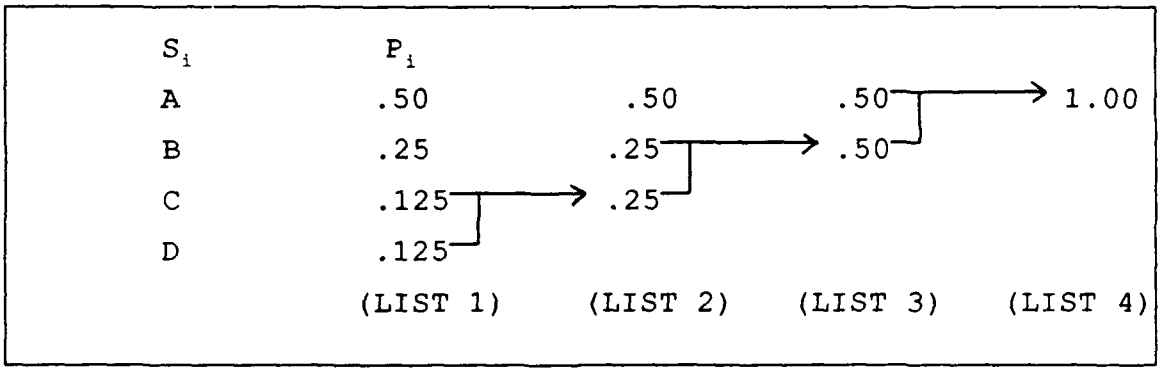


Figure 4.8. Derivation of Huffman Codes by List Method.

- Convert the diagram shown above to a tree structure, maintaining the relative location of the arcs such that of the two arcs entering a node, the upper arc is drawn from the node having the greater probability and the lower arc comes from the node having the lesser probability. If the two nodes which are combined to form a new node have equal probability, then their relative location does not matter. Remember that each time a new node is formed, it is always the current two lower probabilities that are joined. The tree diagram shown in Figure 4.4 shows the tree structure derived from the diagram in Figure 4.8. This example produces a simple tree since no transpositions of probabilities occur. Alternate visual representations by Held [Hel87: p.97] and Davis [Dav88] are shown in Figure 4.9, (A) and (B), respectively.

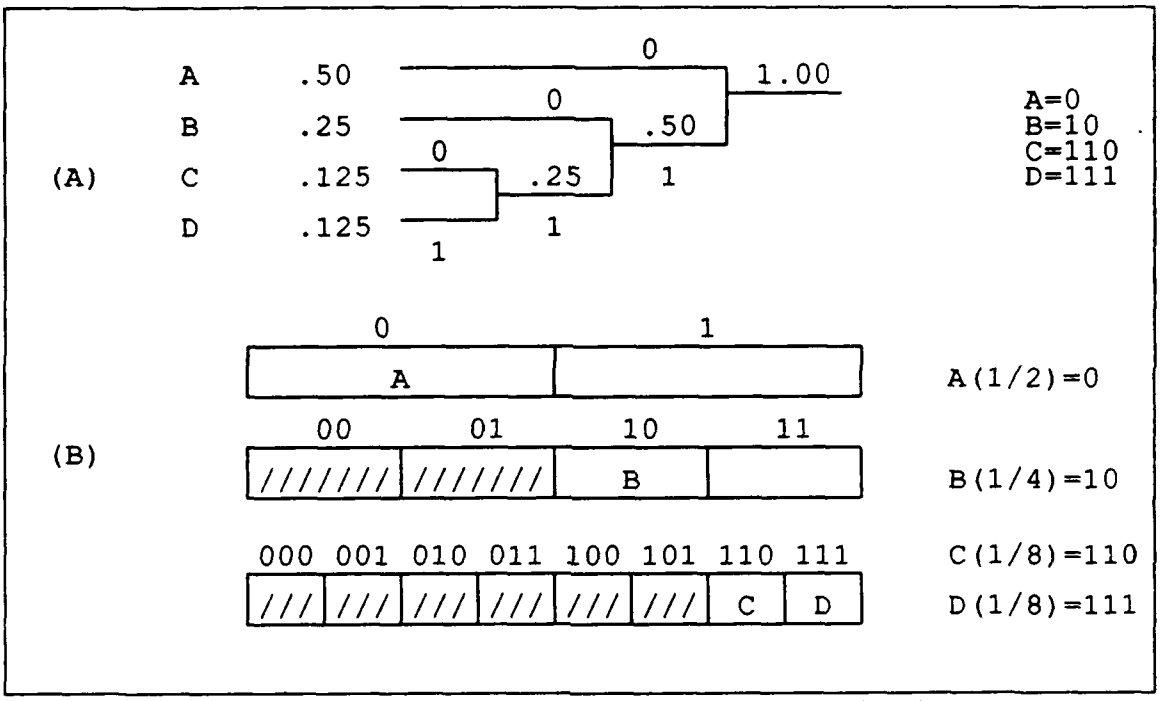


Figure 4.9. Derivations of Huffman Codes (Held and Davis Diagrams).

- Assign a "0" to each upper arc of the tree, and a "1" to each lower arc.
- Beginning at the root node, follow each path back to a leaf node, recording the label of each arc encountered along the way. Each bit string thus derived is the encoded value for that particular source symbol. For instance, following the path to symbol C we record the labels "1," "1," and "0;" thus the encoded value of symbol C is the three-bit string "110."

Step 3. The next step in the process is to encode the message, using the codes for the source symbols derived in Step Two. This is a straightforward substitution encoding process, producing a compressed file as output. For example, source symbols "ABBADAAC" would be encoded as code symbols "01010011100110".

Step 4. Next the encoded message is transmitted. If a table of codes was derived as just described, then this information must also be sent with the message. However, if a predefined frequency distribution of probabilities for the source alphabet was used in the coding process, then the receiver may either re-create the Huffman codes on the receiving computer, or may already have a Huffman coding table resident

on that computer. When the table must be passed with the message, the effect on the size of the transmission due to the table size must be taken into account in determining a valid compression ratio.

Step 5. The final step is to decode the encoded message. It is in the decoding process that the concept of "unique decodability" is fully appreciated. As each variable-length code is received, it can be conclusively associated with one and only one code from the decoding table.

Thus decoding is easily accomplished in one or two passes, depending on the need to derive a decoding table.

2. Another Huffman Code Example

Now that we understand the process of deriving Huffman codes, let us look at a more complicated example. Assume that the data file, i.e., the message, is a color graphics bitmap, and each pixel is represented by three bits. Scanning the data produces a frequency distribution of the colors. Of the eight available colors (the source alphabet), only seven are actually used (source symbols). Figure 4.10 shows the source symbols, the probabilities of occurrence, and the list reordering process. Notice the

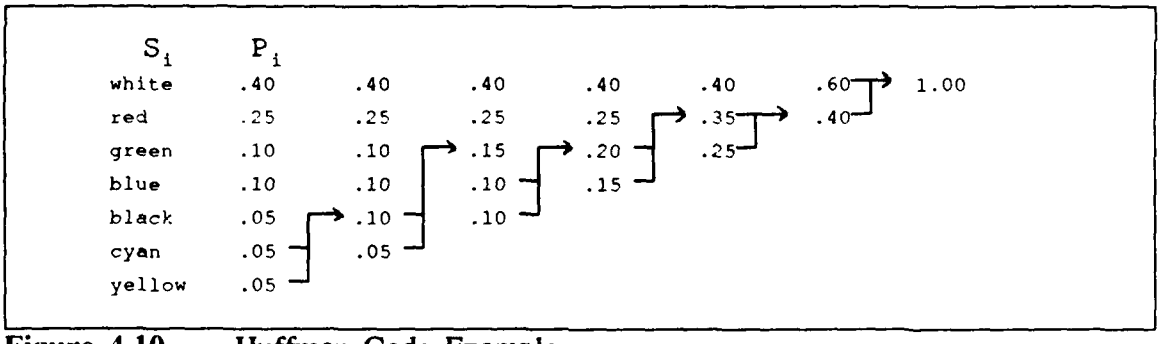


Figure 4.10. Huffman Code Example.

transpositions which occur during this process.

Next, using Held's visual representation of a tree structure, we construct the tree from which to read the code symbols. This construction is shown in Figure 4.11.

The minimum average code length can be calculated for this example from the formula $\sum P_i L_i = 2.40$. We compare this to the calculated value for entropy to estimate the optimality of our derived code $\sum -P_i \log_2 P_i = 1.79$.

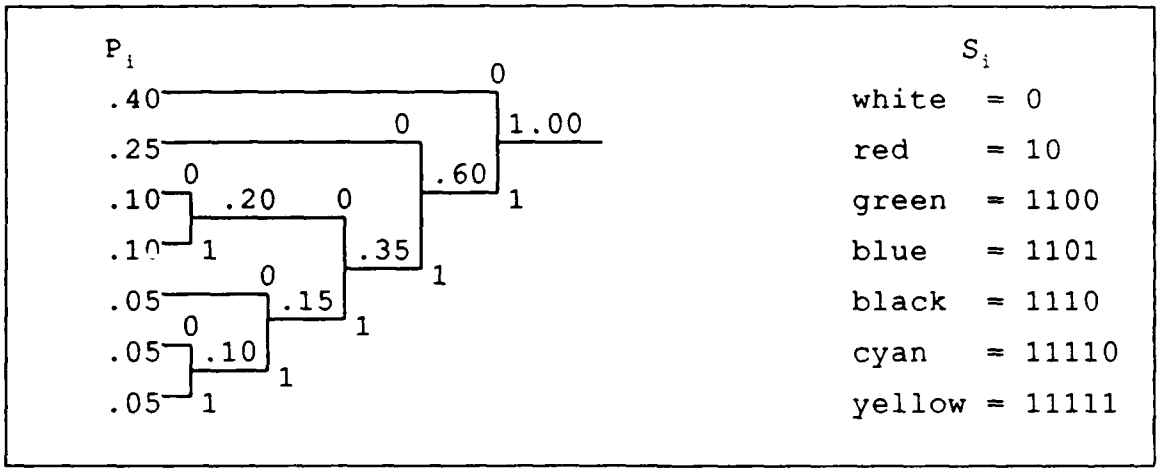


Figure 4.11. Huffman Code Example (Tree Diagram).

Without compression the message requires 3 bits per symbol; with compression it requires an average of 2.4 bits per symbol, a savings of .6 bits per symbol. If we consider, for example, an EGA bitmap with a resolution of 640 by 350 pixels, then the compressed file is 134,400 bits versus 224,000 bits, a savings of 20%.

3. Dynamic Huffman Codes

The Huffman codes which have been discussed so far are those which are either developed by assessing the frequency of the elements used in the message, or chosen from existing tables. Such tables are static, predetermining the probability, and thus the order, of the source symbols. Both the sender and the receiver have on hand identical tables prior to the transmission of the actual message.

It is also possible to dynamically develop Huffman tables. A good description of dynamic Huffman coding is given in the following quotation.

> In a dynamic Huffman model, a frequency algorithm determines which characters are represented at which levels in the table. Every time a character is used, its position in the table is exchanged for the position of the character immediately above it. The bit patterns in the table themselves do not actually change. What changes is the assignment of the bit patterns within a table entry to represent a particular character. An exchange is always made after the code currently assigned is sent across the line. This ensures that both sender and receiver can update their respective copies of the table in synch. [Bac88: p.77,78]

An example of a situation where it would be advantageous to use the dynamic-table method is a text data file which contains lengthy sections of all

uppercase characters interspersed with uppercase and lowercase text. The frequency distribution of the normal text would probably give all uppercase letters very low probabilities. However, in the switch to uppercase letters only, it would be desirable to represent these characters with the shorter codes reserved for frequently used characters. Developing a new table dynamically would allow the uppercase letters to rise to the top of the table, and to be encoded in a shorter form.

V. RELATIVE ENCODING

Numerous other lossless compression techniques exist, including the Lempel-Ziv method, predictive encoding, adaptive encoding, and relative encoding. Many methods are a combination or modification of techniques already mentioned. Telcor Corporation, for instance, claims to have developed a method which combines elements of Huffman and Lempel-Ziv methods, but produces greater compression than either [Bac88: p.77].

A. DESCRIPTION OF RELATIVE ENCODING

We shall examine one of these methods, the relative encoding technique, which is used to transmit data over facsimile devices. This is of interest because of the similarities of this type of data to a graphics bitmapped image. Facsimile data typically is transmitted as 1728 points or pixels per scan line with approximately 850 scan lines for a standard 8½ by 11 inch page.

Relative encoding takes advantage of the fact that facsimile images generally contain a much higher quantity of white space than black space. There may be little change from one scan line to the next. This method transmits only the difference between scan lines. The process of encoding a file by relative encoding is described.

- Read the first scan line into a buffer in memory. Transmit this line exactly.
- Read the next scan line of the file into a second memory buffer. Compare this line to that in the first memory buffer. Transmit only the location of the pixel where a change occurs.
- Move the scan line in the second buffer to the first buffer.
- Continue to execute the two previous steps until all the scan lines of the file have been compared and differences have been transmitted.

This process will become clearer with an example. The asterisks represent the positions where a change has occurred from the top line to the next line.

```
000011111001000000000...00111   Nth scan line
000011110000111100000...00001   (N+1)th scan line
      *  *****        **        Relative change
```

B. HOW TO DESIGNATE RELATIVE CHANGE

There are two ways to designate the relative change or the difference shown from one scan line to the next: "positional notation" or "displacement notation."

1. Positional Notation

A positional indicator may be used for each relative change to indicate the location of the pixel where a change occurs; the location is the number of the pixel changed, relative to the first pixel of the scan line. In our example above, the transmission for the (N+1)th scan line would indicate only the locations of the changed data, i.e., 9, 12, 13, 14, 15, 16, ..., 1726, 1727.

If each digit in a scan-line sequence is transmitted in a four-bit nibble, i.e., digits are packed two to a byte, then greater compaction results than from using a byte to contain the same data. One alternate technique of positional notation allows for the transmission of a positional indicator plus a count of the number of successive changes. Using this method, the above example would result in the transmission of these values: 9, 1, 12, 5, ..., 1726, 2, which further increases compaction.

2. Displacement Notation

Another method for reducing the number of digits required to indicate change is to employ displacement notation. Because a facsimile scan line is long (1728 pixels), changes near the end of a line require four digits as opposed to one or two at the beginning of the scan line. To alleviate this end-of-line increase of digits, the actual location of the first change of a scan line is transmitted, but successive changes in the same line are transmitted as a displacement from the previous change location. Assuming that the change previous to that shown for location 1726 was at location 1716, the above example would be transmitted as 9, 3, 1, 1, 1, 1, ..., 10, 1.

The alternate method of representing changes which are adjacent by a count of the successive changes applies for displacement notation as well. The above example translates to a transmission of 9, 1, 3, 5, ..., 10, 2.

VI. EVALUATION

We have discussed techniques for compressing data using run-length encoding, statistical coding, and relative encoding. Since our interest is primarily in the effectiveness of these methods for compressing graphical bitmapped data, we shall analyze each in terms of (a) binary (black and white) images requiring one bit per pixel, and (b) grey-scale and color graphics images, comprising more than one bit of information per pixel. Binary graphics images, which are analyzed on a bit-by-bit basis, sometimes require a different type of processing from grey-scale and color images, which are analyzed one or more bytes at a time.

A. RUN-LENGTH ENCODING

Run-length encoding can be used on any type of graphical image. The key questions to ask are:

- Is there enough repetition in the file to warrant encoding?
- What is the minimum run length for this file?

Remember that repetition may refer to runs of identical pixels or runs of repeated patterns of pixels. Also recall that typically the minimum run length decreases as the number of bits per pixel increases.

The run-length encoding of a binary image file is most efficiently performed by using a method described in the previous chapter, i.e., by encoding the entire file where each run is represented by one byte, such as an ASCII character selected from a lookup table. But even though each pixel in a binary image file occupies only one bit, encoding a run may require eight or more bits, with a minimum run length of nine pixels. By comparison, an eight-bit grey scale pixel may be encoded in two bytes, with a minimum run length of three pixels.

What are the best, worst, and average amounts of compression attainable by run-length encoding a bitmapped graphics file?

For all bitmapped files, the best compression is on a file which has only one value, and is thus composed of one very long run. The worst compression would be

found in a file which contains no runs of length greater than or equal to the minimum run length. Compression is 0.0% since to encode such a file would create an encoded message larger than the original message. And the average case could fall anywhere in between the extremes. But with any type of run-length encoding, the more repetition the data contains, the more successful the encoding; the longer the runs, the greater the compression.

1. **Best Case -- Binary**

For a binary bitmapped file, the method of compression determines the maximum run length that can be carried in one byte, and thus has an effect on the maximum degree of compression attainable. In Chapter II, were described three methods for compressing this type of file. Each of these methods assumes that the entire file is encoded, as in the best case examined here. Methods One and Two both give 93.7% compression for a high-resolution (640 by 350) EGA bitmap. Method Three gives 99.9% compression for the same resolution. An explanation of these compression calculations follows.

An EGA bitmap occupies 224,000 bits, or 28,000 bytes, of memory. Method One carries the run value in bit one and the run length in the remainder of the byte, for a maximum run length of 127 bits encoded in each byte. Thus 1764 bytes are required to encode a bitmap having only one value, i.e., one run. Method Two carries a maximum run value of 255 bits encoded in each byte; but each byte of encoded data must be separated by a "null" byte encoding a run of zero for the opposite run value. It takes 879 bytes, doubled, or 1758 bytes to encode the bitmap. Method Three, on the other hand, uses a base-128 mode of encoding long runs. Two bytes encodes a maximum run of 16,511 bits; 28 bytes encodes the entire bitmap. This yields a compression factor of 1:1000!

2. **Best case -- Color**

For an explanation of color or grey-scale compression in the best case, assume the entire file is encoded. Thus only two values are needed: one byte to hold the run length (maximum value is 255 pixels) and one or more bytes to hold the run value, depending on the number of bits required to encode one pixel.

The length of the run value may actually be disregarded in calculating compression. Since it is true that every run of 255 pixels may be encoded by one pixel value plus a one-byte run length, then compression is approximately 2:255. But to be more specific, if a pixel is contained in eight bits, then every 255 bytes of data is encoded with two bytes of data yielding a compression of 99.3%. A 32-bit pixel value means that 255 pixels, or 1020 bytes of data, are encoded with five bytes, for a compression of 99.5%.

B. STATISTICAL / HUFFMAN CODES

In evaluating the effectiveness of statistical encoding, it is important to remember the premise on which use of these codes is based. Remember that statistical codes are variable length, whereas run-length codes are fixed length. The main idea inherent in statistical coding is that symbols which occur more frequently than others in a message will be replaced by shorter codes, while symbols which are used less frequently will be replaced by longer codes. The concept of entropy, a measure of "surprise" at the occurrence of a symbol in a message, is used as the best value, for the average bit count in a specific encoding.

We examine the best, worst and average case for statistical coding in general. The conclusions also apply to Huffman codes. In this evaluation, we do not distinguish among binary, grey-scale, or color bitmaps. A binary bitmap must be considered in blocks of pixels, or patterns, because it makes no sense to encode an alphabet containing only two symbols by the statistical method. To do so would create a mapping of the source symbols "1" and "0" onto the code symbols "1" and "0" respectively.

1. Best Case -- Statistical Codes

The formula for calculating entropy is $-\sum P_i \log_2 P_i$, where P_i is the probability that the i^{th} symbol in a message will occur. The lower the entropy, the smaller the average number of bits required to derive an encoding. The best situation, therefore, is the data file which is composed entirely of one character, or one bit pattern. There is no surprise as to which symbol will be received next in a transmission. The probability of receiving the particular symbol used is one and the entropy is zero! This

means that each occurrence of the symbol in the message can be encoded with one bit. If the symbol is an eight-bit character or pixel, then compression is 1 - 1/8 = 87.5%. If the symbol is a 32-bit pixel, then compression is 96.9%. Both of these upper compression limits hold, regardless of the size of the bitmap.

2. Worst Case -- Statistical Codes

Since the philosophy of statistical encoding is to reduce the average number of bits per character by using short codes for highly probable pixel representations, we may presume that the advantage will be lost if every character, or pixel value, occurs with equal probability. We have seen that one pixel value produces maximum compression, and no surprise. Let us look at more than one symbol, transmitted randomly, and observe the entropy, or surprise factor, for these situations. Figure 6.1 does this. It can be seen from the examples in the figure, that if the selected number of source symbols in a file is 2^n, then the entropy, or best expectation for the average number of bits per pixel, is n. It can also be seen that as the number of symbols increases, so does the average size of an encoded pixel. Assuming an eight-bit pixel

Number of Symbols	Probability Symbol Will Occur	Entropy
1	1.00	0
2	.50	1
4	.25	2
8	.125	3
16	.063	4
32	.031	5
64	.016	6
128	.0078	7
256	.0039	8
512	.0019	9

Figure 6.1. Symbols Occurring with Equal Probability in Graphics Data Encoded with Statistical Method.

representation, it is clear that for more than 128 source symbols with equal distribution, there is no benefit to using statistical encoding. This represents the worst case.

3. Average Case -- Statistical Codes

It is difficult to derive a generalized "average" compression value for statistical encoding. In Chapter VII, where we look at one method of using Huffman codes, about 40 actual binary bitmaps were encoded and an average compression value was computed for that particular sampling. Because the use of graphics images is so varied, ranging from binary satellite map data to multicolor molecular diagrams, sampling each particular situation seems a reasonable method of deriving an average compression value.

Another consideration in evaluating statistical encoding is the overhead incurred by the encoding method. In Huffman encoding, for example, if the static method is used, the program must make two passes of the data to first calculate the frequency distribution of the symbols, and then to encode the file. Time to transmit the lookup table must be considered, in addition to the I/O time to read the file twice and the computing time for encoding. Use of a dynamic Huffman method eliminates one pass of the data file and the transmission of the lookup table, but increases the compute time required by both host and target machine in order to remain synchronized.

C. RELATIVE ENCODING

Relative encoding relies on the supposition that bits in any given scan line of a bitmap differ little from the previous line. This assumption implies the existence of vertical patterns. Once the first line of the bitmap is transmitted, only the relative differences of subsequent lines are transmitted.

With this in mind it is easy to see that the maximum compression is obtained when every line of the bitmap is identical. Minimum compression occurs for a bitmap where every bit in a scan line is the opposite of the bit immediately above it in the previous line. An average compression would fall between these two extremes.

Although relative encoding is used in facsimile transmissions and may lend itself well to compression of a binary image bitmap, it may not prove a satisfactory method

for compressing grey scale or color bitmaps due to the greater possibility of vertical variation in the map. The advantageous situation occurs in a graph that has large filled areas of one shade, or any combination of pixels which provides very little vertical change.

The important issue in any encoding scheme is that the host machine and the target machine maintain synchronization, so that at each transmission, the target machine knows exactly how to interpret and decode the encoded message being sent.

VII. A RUN-LENGTH / HUFFMAN IMPLEMENTATION

This chapter describes an existing program, GRAFPC, which is currently used by computer users at the Naval Postgraduate School to transfer graphics files created on the IBM 3033 mainframe computer to an IBM-compatible microcomputer, and to view them on the PC monitor. First, the method of run-length compression currently implemented within the program is described. Next an analysis of a proposed implementation of Huffman coding "on top of" the run-length encoding is presented. Finally, the chapter concludes with a discussion of methods to further compress data from the GRAFPC program.

A. DESCRIPTION OF THE GRAFPC PROGRAM

1. Background

GRAFPC was written by Mike Gunning, while a staff member of the Meteorology Department at the Naval Postgraduate School, to fill a need of students who owned microcomputers, could link (via SIM/PC™ [3]) to the mainframe computer and execute DISSPLA to create graphics, but who could not view the results on their computers at home. SIM/PC is an asynchronous communications package which provides micro to mainframe connectivity. CA-DISSPLA™ [4] is a sophisticated graphics program which executes on the IBM 3033 mainframe computer at the Naval Postgraduate School. GRAFPC was designed to enhance the output capabilities of DISSPLA. [Gun84]

---

[3] SIM/PC is a proprietary product of SimWare Corporation.

[4] CA-DISSPLA is a proprietary product of Computer Associates, Incorporated.

2. How GRAFPC Works

GRAFPC consists of two distinct parts. The first part is the output device driver which resides on the host mainframe computer, embedded in DISSPLA. This Fortran program converts the user's graphic to a bitmap, compresses it with run-length encoding, and creates an ASCII file which can then be transmitted by SIM/PC from the mainframe to a microcomputer.

The second part is an assembler language "terminate-and-stay-resident" program which resides on the target microcomputer. This program filters the data stream transferred by SIM/PC, waiting for a "start of plot" sequence ("ll"). Upon receiving this prompt, GRAFPC captures all encoded data, through the "end of plot" character ("~"), decompresses it, and stores the reconstructed bitmap into the computer's video memory area.

See Figure 7.1. for a diagram of this process. It is the first part of

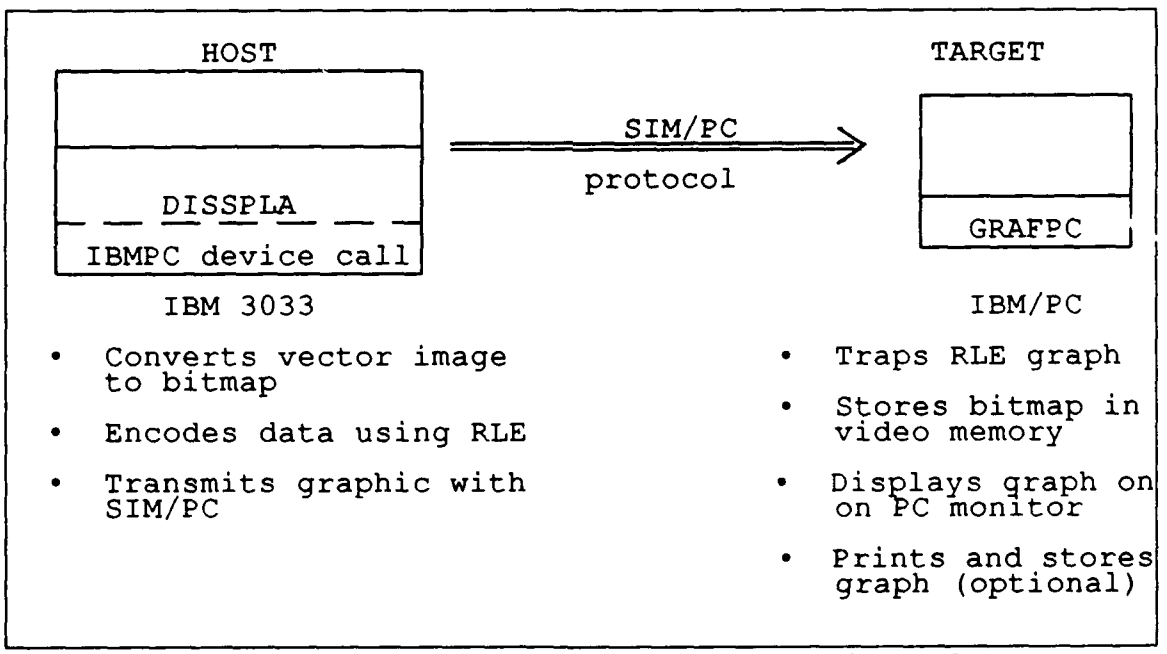


Figure 7.1. The Process of Transmitting a Graph Using GRAFPC.

GRAFPC with which we are concerned, the Fortran device driver residing in DISSPLA on the mainframe computer. This is where the graphic data is converted to a bitmap and compressed in preparation for transmission to the microcomputer.

3. Design Considerations

Several problems were encountered with the initial methods of transferring a GRAFPC graphic. These concerned the issues of conversion and size of data.

Because the host IBM computer uses the EBCDIC character set and the target microcomputer uses ASCII, sending an unformatted bitmap resulted in lost or garbled characters caused by conversion problems. Formatting the bitmap and sending each eight bits (one byte) of data as an integer (using Fortran "I3" format) worked, but created too large a file. A 28 kilobyte file (the bitmap size of EGA, for instance) would triple to 84 kilobytes.

Since every graph has much white space and originally exists in DISSPLA in a vector format, the idea of transmitting the vectors was considered. However, there could be 30 or 40,000 vectors in a single graph, each requiring that a pair of coordinates be transmitted. At least the size of a bitmap is consistent, whether the graph contains thousands of vectors (as is true in using shaded characters) or very few.

Because of these problems, it was decided that compression of bitmapped data, using characters common to both EBCDIC and ASCII, would be used in GRAFPC.

B. COMPRESSION BY RUN-LENGTH ENCODING IN GRAFPC

Prior to transmitting a bitmapped graphic, GRAFPC compresses the data using run-length encoding. Since it was desirable to compress the binary bitmapped file in such a way that bytes of character data could be transmitted, the third method of run-length encoding described in Chapter III of this thesis was a likely candidate.

A list of 91 characters common to both EBCDIC and ASCII was selected. Figure 7.2. depicts the resulting "lookup table" used for this procedure. Identical tables exist in the Fortran device driver software of DISSPLA and in the assembler software of GRAFPC.

The transmitted file consists of a **metacode** of run-length characters from the lookup table, embedded between the "start-of-plot" sequence ("!!") and the "end-of-plot" character ("~"). An appropriate number of blank characters are sent to ensure proper

```
 0 - !     15 - 0    30 - ?    45 - N    60 - '    75 - o
 1 - "     16 - 1    31 - @    46 - O    61 - a    76 - p
 2 - #     17 - 2    32 - A    47 - P    62 - b    77 - q
 3 - $     18 - 3    33 - B    48 - Q    63 - c    78 - r
 4 - %     19 - 4    34 - C    49 - R    64 - d    79 - s
 5 - &     20 - 5    35 - D    50 - S    65 - e    80 - t
 6 -Z7D    21 - 6    36 - E    51 - T    66 - f    81 - u
 7 - (     22 - 7    37 - F    52 - U    67 - g    82 - v
 8 - )     23 - 8    38 - G    53 - V    68 - h    83 - w
 9 - *     24 - 9    39 - H    54 - W    69 - i    84 - x
10 - +     25 - :    40 - I    55 - X    70 - j    85 - y
11 - ,     26 - ;    41 - J    56 - Y    71 - k    86 - z
12 - -     27 - <    42 - K    57 - Z    72 - l    87 - {
13 - .     28 - =    43 - L    58 - \    73 - m    88 - |
14 - /     29 - >    44 - M    59 - _    74 - n    89 - }
                                                   90 - ~
```

Figure 7.2. Lookup Table for Run-Length Encoding in GRAFPC.

alignment of the metacode within the CMS environment of SIM/PC. Following is a description of the run-length encoding method employed in GRAFPC.

This scheme is based on the assumptions that (a) the entire bitmap will be encoded, (b) the first value of the metacode will represent the length of a run of "on" bits, and (c) each line of the bitmap will be encoded separately, using an "end-of-line" character ("}") to signify this condition. Thus each character transmitted represents a run of either "bits on" or "bits off."

Of the 91 compatible characters in the table, three are designated for marking the beginning of the plot, the end of a line, and the end of the plot; therefore, only 88 characters are available for indicating a run length.

One line of a bitmap may contain over 700 bits, often all of the same value, so a base-80 encoding scheme is used to allow for so-called "long runs." The first 80 characters indicate actual run lengths of zero through 79. For runs greater than 79, two-character encodings are used. The first character has a value from 80 through 87, but represents a multiple of 80; the second character has a value from zero through 79. As in any n-base number system, the formula used to compute the run value is

((<first character>-79) * 80) + <second character>.

An example shows two run lengths: the first is a run less than 80, and the second is a long run.

encoding: Av8

meaning: "A" represents a run of 32 bits

"v8" represents a run of (82-79) * 80 +8 = 248 bits

Figure 7.3. shows a sample graph produced by DISSPLA, as well as the accompanying compressed metacode output which is transmitted by GRAFPC from mainframe to microcomputer.

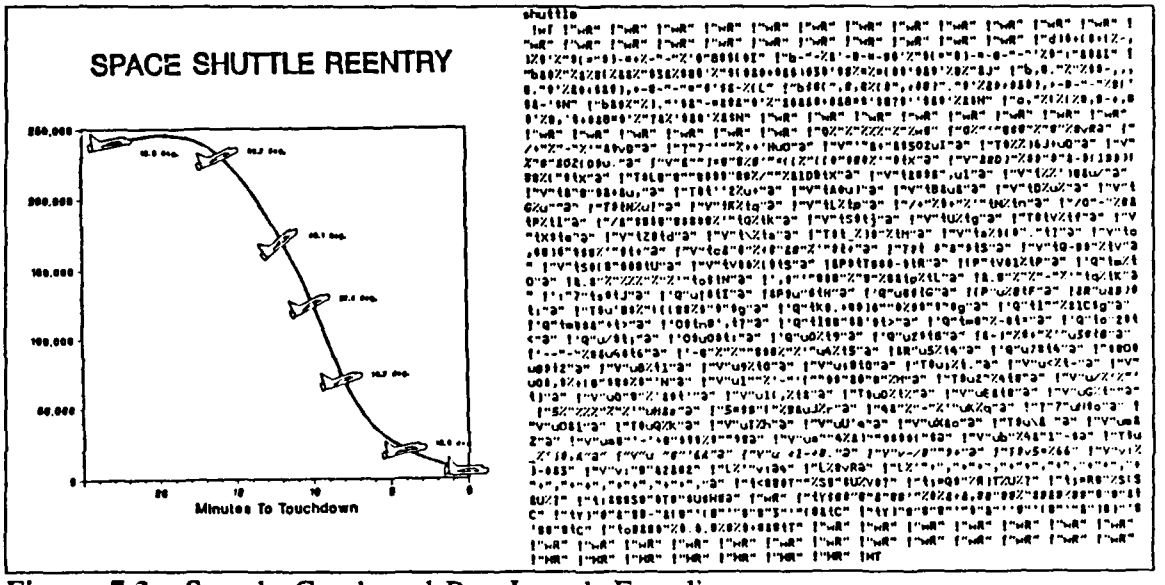


Figure 7.3. Sample Graph and Run-Length Encoding.

C. USE OF HUFFMAN CODES TO IMPROVE COMPRESSION

In spite of the acceptable compression obtained on a bitmap by using run-length encoding, the time to transmit a graph via GRAFPC is generally about one minute, a long time to wait for results. On a sample set of graphs, the average compression from the run-length encoding is 57%.[5] In an effort to reduce the transmission time, it was decided to study the effect of further compressing the metacode file by using a Huffman encoding method on the run-length encoded data.

[5] The sample subset includes two graphs whose run-length encoding exceeds the size of the original bitmap. Without these graphs, the average compression is 61.7%.

This section describes the goals and the design of the implementation of such an experiment and shows the results obtained. It should be noted that compression of the RLE metacode by a Huffman encoding is not fully implemented. Only the program coding necessary to collect the statistics on compression obtainable by such a technique has been written. The doubly-compressed file is neither created nor transmitted.

1. Goals of the Implementation

The goals are two-fold. Of primary importance is a determination of the amount of compression we achieve by using the Huffman encoding technique on an RLE metacode file. And secondly, how effective is this technique? Section B of Chapter Two provides an elaboration of the concepts mentioned below.

In analyzing compression, it is of interest to look at (a) the compression originally achieved by the run-length encoding method, (b) the additional compression achieved by encoding this file with Huffman codes, and (c) the total compression achieved by the double compression method.

Before performing such an analysis, it is necessary to define the data which is needed for the results. The information necessary to compute the compression results includes the original bitmap size, the total number of characters in the RLE metacode, and the average size of a Huffman encoding for each particular graph considered. Since the average length of a Huffman code for a given graph is calculated by the formula

$$L_{avg} = \sum P_i L_i$$

it can be seen that both the probability of occurrence, as well as the length of each source symbol used in the RLE metacode of a graph, must be known. The Huffman encoding must first be performed in order to arrive at the average code size.

One measure of the efficiency of the Huffman code obtained for a graph is the amount of **redundancy** present in the encoding [Lel88: p.267]. Redundancy is defined as the difference between the average code length of the encoding method (in this case, the Huffman method) and the **entropy**, or average information content, of the encoding method. The formula for measuring redundancy is

$$\sum P_i L_i - \sum -P_i \log_2 P_i.$$

2. Choosing Representative Graphs for Testing

The next step was to select a sampling of graphs on which to generate the appropriate data for analysis. The following criteria were used for choosing this subset of graphs.

- The sample set should produce a range in the number of source symbols used in the RLE metacode.
  Rationale: If a graph contains many runs of the same length, the number of symbols required will probably be small; runs of different lengths provide more symbols in the encoded file.
- The sample set should provide a range in the size of the encoded file.
  Rationale: If the graph is simple and good compression is obtained, the RLE metacode file will be relatively small; it will be larger for poorly compacted graphs.

Graphs to be included in the sampling were chosen by trial and error. Except for the most elementary graphs, most seemed to fall at the high end of the ranges mentioned.

Another question to consider is the relationship of output resolution to degree of compaction. GRAFPC was written to produce bitmaps of four resolutions, for CGA, EGA, color-400, and Hercules monitors. What is the effect of the double compression on the same graph produced at different resolutions?

3. Design of the Implementation

The question is "How much compression can we obtain if we further compress the RLE metacode by encoding it by the Huffman method?" The technique used to determine the answer to this question involves the following steps:

Step One. Generate a file containing RLE metacode.

Step Two. Determine the subset of characters used in this file from among the source alphabet of 91 characters.

Step Three. Calculate the frequency distribution of these characters within the metacode file, i.e., "What is the percentage of use of each character?" (Pass One)

Step Four. Construct a Huffman code for each character in the subset. (Pass Two)

Step Five. In order to analyze the efficiency of the Huffman encoding, calculate the entropy (smallest expected number of bits per character for this subset),

the average number of bits per character for the Huffman coding, and the redundancy of this encoding.

Step Six. Determine the overall compression obtained by doubly compressing the original graphics bitmap.

Appendix B contains illustrations of the graphs in the sample subset. Credits for the originators are included. A listing of the program which generated the compression data is shown in Appendix C. And Appendix D shows sample output from one graph, including RLE data, the Huffman codes generated, and compression analysis data.

4. Results

Figure 7.5. shows the results of executing the above six-step program

	SIZE	SYMBOLS	COMPACT	FACTR	LAVG=	ENT=	REDUN
3curves	7003	89	41.37	1.7	4.69	4.66	0.03
simpcrv	3843	88	42.56	1.7	4.60	4.56	0.03
interp	9061	86	42.65	1.7	4.59	4.53	0.06
shuttle	7008	89	42.74	1.7	4.58	4.55	0.03
contour	11585	88	43.75	1.8	4.50	4.46	0.04
t3d	10280	88	44.50	1.8	4.44	4.42	0.02
shapes	11197	80	44.90	1.8	4.41	4.38	0.03
funplot	6273	86	44.95	1.8	4.40	4.36	0.04
dream	7968	88	45.17	1.8	4.39	4.35	0.04
usamap	10492	88	45.76	1.8	4.34	4.30	0.03
hrt	4709	87	46.21	1.9	4.30	4.26	0.05
gas	9861	88	46.56	1.9	4.28	4.24	0.04
wldplt	12881	88	46.69	1.9	4.26	4.23	0.03
mapgrid	9783	87	46.79	1.9	4.26	4.21	0.05
spiral	5486	83	47.09	1.9	4.23	4.17	0.06
gantt	6995	86	47.64	1.9	4.19	4.17	0.02
k2	5452	87	47.94	1.9	4.16	4.12	0.04
eztest	6781	88	48.62	1.9	4.11	4.07	0.04
bar	6762	86	49.40	2.0	4.05	4.02	0.03
grids	10541	78	49.86	2.0	4.01	4.00	0.01
sorrento	12975	88	50.05	2.0	4.00	3.97	0.02
mount	12125	89	50.18	2.0	3.99	3.96	0.03
2pies	16912	86	50.70	2.0	3.94	3.90	0.04
steffin	7808	86	52.88	2.1	3.77	3.72	0.05
gday	8269	80	53.39	2.1	3.73	3.67	0.06
targt	19073	86	54.08	2.2	3.67	3.62	0.06
pie	14897	88	54.40	2.2	3.65	3.61	0.04
plotext	14948	84	54.93	2.2	3.61	3.55	0.06
shdcrv	21477	89	55.28	2.2	3.58	3.51	0.06
thred6	20995	86	58.23	2.4	3.34	3.30	0.04
atombox	25321	82	61.31	2.6	3.10	3.05	0.04
logcrv	33750	72	63.02	2.7	2.96	2.94	0.02
cpi	21022	86	63.29	2.7	2.94	2.86	0.08
gridlog	37043	78	63.36	2.7	2.93	2.91	0.02
eframe	1990	7	68.72	3.2	2.50	2.27	0.24

Figure 7.4. Results of Huffman Coding the RLE Metacodes from a Sample Set of DISSPLA Graphs. (EGA)

against the sample set of graphs. The output is for graphs transmitted in CGA resolution. Figure 7.4. shows output of the same subset of graphs transmitted in EGA resolution. The data in the charts is sorted according to the amount of compression obtained by encoding the RLE metafile.

	SIZE	SYMBOLS	COMPACT	FACTR	LAVG=	ENT=	REDUN
3curves	3979	89	38.96	1.6	4.88	4.85	0.03
shapes	5298	80	39.43	1.7	4.85	4.82	0.02
simpcrv	2200	87	40.95	1.7	4.72	4.69	0.03
interp	5143	86	41.06	1.7	4.72	4.66	0.05
t3D	5499	88	42.57	1.7	4.59	4.57	0.02
funplot	3648	85	43.01	1.8	4.56	4.53	0.03
contour	6906	87	43.20	1.8	4.54	4.51	0.04
usamap	5854	87	43.80	1.8	4.50	4.47	0.02
dream	4563	87	43.82	1.8	4.49	4.47	0.02
wldplt	6531	87	43.97	1.8	4.48	4.46	0.02
mapamer	4297	87	44.27	1.8	4.46	4.43	0.03
k2	1842	81	44.82	1.8	4.41	4.40	0.02
heart	2647	84	44.84	1.8	4.41	4.38	0.03
shuttle	2855	85	44.85	1.8	4.41	4.39	0.02
mapgrid	5582	87	45.53	1.8	4.36	4.33	0.03
spiral	3048	79	45.69	1.8	4.35	4.31	0.04
gas	5807	88	45.96	1.9	4.32	4.29	0.04
gantt	4076	83	46.30	1.9	4.30	4.27	0.02
sorrento	6640	88	47.16	1.9	4.23	4.21	0.02
grids	5729	74	47.42	1.9	4.21	4.17	0.04
mount	6595	89	47.51	1.9	4.20	4.17	0.03
2pies	9301	86	48.51	1.9	4.12	4.08	0.04
pie	7588	87	49.87	2.0	4.01	3.97	0.04
steffin	4507	73	49.91	2.0	4.01	3.94	0.06
threD6	9417	86	50.81	2.0	3.93	3.91	0.03
shdcrv	8039	82	51.11	2.0	3.91	3.86	0.05
plotext	4144	81	51.32	2.1	3.89	3.85	0.04
targt	10002	82	52.08	2.1	3.83	3.81	0.03
Gday	5444	81	54.22	2.2	3.66	3.60	0.07
atombox	12466	79	58.23	2.4	3.34	3.33	0.01
bar	6446	55	60.80	2.6	3.14	3.09	0.05
testplot	6448	64	61.06	2.6	3.12	3.07	0.04
cpi	11866	81	62.07	2.6	3.03	2.97	0.07
logcrv	18350	67	62.23	2.6	3.02	3.00	0.02
gridlog	20069	74	63.38	2.7	2.93	2.91	0.02
eframe	1126	8	68.59	3.2	2.51	2.31	0.20

Figure 7.5. Results of Huffman Coding the RLE Metacodes from a Sample Set of DISSPLA Graphs. (CGA)

SIZE is the number of symbols used in the message. SYMBOLS is the number of source symbols actually used, from a maximum of 90[6]. COMPRESS is the amount of compression the Huffman coding provides and is calculated by the formula:

[6] The "start of plot" character ("||") is not encoded as it must be recognized by GRAFPC on the microcomputer.

1-(AVG/8). FACTR is the compression factor, or the ratio of uncompressed data (eight bits) to compressed data (AVG). AVG is the average number of bits required to encode a source symbol (an RLE character). ENT is the entropy for the particular graph. And REDUN shows the redundancy, the difference between ENT and AVG.

One goal of a double compression of GRAFPC bitmaps is to analyze the efficiency of the Huffman coding. This is measured, as shown in Figures 7.4 and 7.5, by the redundancy. In most cases, the average symbol size of the Huffman codes is extremely close to the entropy, the best expected average symbol size. These results indicate that using the Huffman technique on the RLE data is a very efficient method of compression.

Figure 7.6. shows a comparison of the graphic data run under two different

AVERAGES:	CGA	EGA
SIZE	6498.70	12359.00
SYMBOLS	79.80	83.30
AVG NUM BITS	4.07	3.96
COMPRESSION	49.15	50.54

Figure 7.6. Average Results of Huffman Coding on RLE.

resolutions: CGA (640 by 200 pixels) and EGA (640 by 350 pixels). Although the higher resolution does provide sightly improved compression, the difference is not significant. Therefore, the double compression is unaffected by the degree of resolution of the bitmap.

Another of the goals is to determine how much compression is obtained by the original run-length encoding, by an additional Huffman encoding, and the total compression derived by both techniques together. Figure 7.7. shows the resulting compression values for the sample set of graphs, using CGA resolution.

NAME	SIZE	%RLE	%HC	TOTAL
3curves	3979	73.68%	38.96%	83.94%
shapes	5298	64.96%	39.43%	78.78%
simpcrv	2200	85.45%	40.95%	91.41%
interp	5143	65.99%	41.06%	79.95%
t3d	5499	63.63%	42.57%	79.11%
funplot	3648	75.87%	43.01%	86.25%
contour	6906	54.33%	43.20%	74.06%
usamap	5854	61.28%	43.80%	78.24%
dream	4563	69.82%	43.82%	83.05%
wldplt	6531	56.81%	43.97%	75.80%
mapamer	4297	71.58%	44.27%	84.16%
K2	1842	87.82%	44.82%	93.28%
hrt	2647	82.49%	44.84%	90.34%
shuttle	2855	81.12%	44.85%	89.59%
mapgrid	5582	63.08%	45.53%	79.89%
spiral	3048	79.84%	45.69%	89.05%
gas	5807	61.59%	45.96%	79.25%
gantt	4076	73.04%	46.30%	85.52%
Sorrento	6640	56.08%	47.16%	76.80%
grids	5729	62.11%	47.42%	80.08%
mount	6595	56.38%	47.51%	77.11%
2pies	9301	38.49%	48.51%	68.33%
pie	7588	49.81%	49.87%	74.84%
Steffin	4507	70.19%	49.91%	85.07%
threed	9417	37.72%	50.81%	69.36%
shdcrv	8039	46.83%	51.11%	74.01%
plotext	4144	72.59%	51.32%	86.66%
targt	10002	33.85%	52.08%	68.30%
gday	5444	63.99%	54.22%	83.52%
atombox	12466	17.55%	58.23%	65.56%
bar	6446	57.37%	60.80%	83.29%
testplot	6448	57.35%	61.06%	83.39%
cpi	11866	21.52%	62.07%	70.23%
logcrv	18350	-21.36%	62.23%	54.16%
gridlog	20069	-32.73%	63.38%	51.39%
eframe	1126	92.55%	68.59%	97.66%
AVERAGES:	6499	57.02%	49.15%	79.21%

Figure 7.7. Results of All Compression Methods on GRAFPC Sample Graphs.

VIII. CONCLUSION

A. WILL DATA COMPRESSION REMAIN IMPORTANT?

In this thesis we have shown the importance of data compression, especially in respect to graphics data. Data which is encoded to require less space, saves storage and reduces transmission time.

One might argue that data compression will become less significant as improved technology facilitates the transmission of greater amounts of data. Two ways this increased information flow is made possible are via faster speed as provided by fiber optic cables, for instance, and by increased bandwidth capacity.

However, as the capability to transmit more data increases, so does the desire (or need) to do so. The dramatic increase in the resolution of computer monitors is a typical example. In the early 1980's IBM introduced the CGA (Color Graphics Adaptor) to display graphics in four colors on the PC monitor at a 320 pixel by 200 pixel resolution. By 1988, the company had introduced the 8514 Display Adapter which can display 256 colors at a resolution of 1024 pixels by 768 pixels. The increased resolution creates video bitmap files which require more memory to store them. Thus the need to compress graphics data, both for storage and for transmission purposes, still exists.

As technological advances provide for faster data transmission and greater storage, new needs will always arise requiring full use of these capabilities. Therefore, the ability to compress data will remain an important area of study. [Eub89]

B. THESIS GOALS REVISITED

As stated in Chapter I, the goals of this thesis are (a) to examine and evaluate several methods of graphic data compression which are used in the field of computer science, and (b) to look at these methods in relation to transmitting graphic images from the IBM 3033 to microcomputers in order to determine a reasonable method of reducing the image transmission time.

In Chapters II through V, we discussed in depth the compression methods of run-length encoding, statistical encoding (including Huffman codes), and other methods such as relative encoding. Each of these techniques was evaluated in Chapter VI.

Chapter VII addressed the question of how to improve the compression achieved by an existing program, GRAFPC, which transmits graphics data from mainframe to PC. Two methods of compression, run length and Huffman encoding, were combined to obtain an average 79% compression on a sample set of graphs, thus providing very successful results.

Another issue is that not all techniques provide equal compression for all types of graphs. There are several points to consider in selecting an appropriate compression technique. One consideration is the **type of data** that is being compressed. For instance, is it character, numerical, or binary bitmapped graphic data?

Another consideration is the **tradeoff in time** required to compress the data and perform error checking versus the time saved by the amount of compression obtained. But as the processors on both the mainframe and PC become faster, the time required for compression operation becomes minimal.

A final issue is the **frequency** with which the data is accessed. Is a particular graph transmitted several times in a session, as in the case of graph development with GRAFPC; or is the data file part of a graphics data base where many graphs are transmitted in seeking a desired final product?

C. IMPLEMENTATION SUGGESTIONS

1. Other Combination Methods

This thesis explored one implementation of graphic data compression, that of combining run-length encoding with the Huffman code method. Other combinations are certainly possible and are suggested as a topic for further exploration. For instance, the original RLE file created by GRAFPC can be doubly compressed by run-length encoding any patterns identified. Or the RLE file can be used as input for a relative encoding implementation. Consider in Figure 8.1 a segment of the RLE data from one of the graphs of the sample set.

```
BAR
}
!wT)
!"wR"}
!"wR"}
!"wR"}
!"wR"}
!"wR"}
!"wR"}
!"wR"}
!"wR"}
!"wR"}
!"wR"}
!"T'"%"#'2"&"'"#',%$'"("###$&:"&"#'#&+"('#'"(""%"$%$'#"%"e";
....

!"t'"%"$%&"-"&""")%-%%%#(""%"$%<%%%%,'#'#")$$"%"$%%%$"%"e"}
!"wR"}
!"wR"}
...same line 10 times

!"uM+#,#($%%%#$"/$,&(&"%(Z"}
!"uL$$"#"&"##$##"%#"#$"###%"#""##""##""##"$""#"'#$#$$%"%#Y"}
!"W&tg%#"&"#(#'"#&""#%"%#%%#"#%$$*($""#$'Z"}
!"W&td$#####$##$##"$$"$"$"$###%%##"%#$"%"'#$##&$"%#Y"}
!"W&"&t\"##&$%(#%##$#'$&%#)$"")&("%")$"Y"}
!"W"$"&"vd"}
!"d"vd"}
!"d"vd"}
!"d"vd"}
....
```

Figure 8.1. RLE File for Graph BAR.

This data is printed so that the encoding of each scan line in the original bitmap is on one line, terminated by the end-of-line symbol ("}"). Seen this way, it is easier to identify the many patterns that exist. Also, the variations from one scan line to another are more obvious; the fewer variations there are, the more a relative encoding scheme compresses the data.

2. Lossy Compression

Methods of lossy compression discussed in Chapter II can be adapted to run under GRAFPC. Of particular interest is the method used by Dr. James Murphy at the University of California at Santa Cruz. This method compresses the data spatially, by transmitting a lower resolution bitmap. [Mur88a] Since the users of GRAFPC are mainly interested in verifying the correctness of their graph development as seen on the PC monitor, the resolution of the transmitted graphic need only be sufficient for this purpose. The final output is generally a printed or plotted graph.

3. Mixture of Methods

A last suggestion for further implementation is to compress parts of a file with different compression methods. Figure 8.1 shows only part of an RLE file. In the complete file for this particular graph ("bar"), there are 25 encoded lines (approximately one eighth of the file) which exceed 80 bytes in length. Since the original bitmap is 640 pixels wide, and each pixel occupies one bit, these lines, encoded, are actually longer than the original source scan line. Would it be better to not encode these lines, or to use a better method just for this portion of the file?

The difficulty in this type of implementation is to identify, dynamically, the compression method to be used for a given part of a file. The burden of the task falls in the area of defining the criteria which will select a given method.

Suggested here is a method by which an entire file may be encoded by any of several available methods, and the decision to encode is made dynamically, at the time of encryption. This logic may be utilized in solving the problem of encoding parts of a file by a choice of methods.

In gathering the statistics for the double compression method used in Chapter VII for GRAFPC, the following observations were made.

- In general, the fewer the number of symbols used from the source alphabet, the better the compression by the Huffman method.
- The RLE encoding of a few of the graphs in the sample set created a file greater than the original bitmap.

Thus, from the sample set data in Figures 7.4 and 7.5, the SIZE and SYMBOL information yield information which may be used to identify, with some degree of confidence, a file which will not compress well with Huffman codes or run-length encoding. Figure 8.2 shows the steps which were used to determine the statistical results of the RLE / Huffman encoding. The process is described in pseudo-code. It should be noted that the SEND routine also exits the program.

This same logic may be used to carry the process one step further, i.e., to dynamically determine, within a data file, what method of compression is appropriate to use for a given segment of a file.

```
run-length encode the bitmap ( => RLE)
    if size of RLE is greater than size of bitmap
        then do;
            relative encode the RLE ( => RELATIVE)
            if size of RELATIVE is less than bitmap
                then SEND (RELATIVE)
            end;
perform frequency distribution on RLE ( => HC)
     if number symbols is less than 90%
          then SEND (HC)
          else do;
               do pattern recognition on RLE ( =>PATTERN)
               if size of PATTERN less than size of RLE
                    then SEND (PATTERN)
                    else SEND (RLE)
               end;
```

Figure 8.2. Pseudo-code to Dynamically Determine Compression Method.

In conclusion we have shown that compression of graphic data is important, methods which have been in use for some time are still valid (e.g., run-length encoding and statistical codes), and that good results can be obtained by combining several methods of compression.

APPENDIX A

GLOSSARY OF TERMS

bitmap - a virtual representation, generally in memory, of a screen image of a target monitor.

compaction - compression; a method of making a file smaller.

compression (more precisely **image compression**) - the act of encoding a graphics file in such a way that it occupies less space in memory.

compression indicator character - a character used to indicate that compressed data follows.

The character chosen should not normally be found in the file; unprintable characters or seldom-used special characters are good candidates. Appearance of the compression indicator character in the original data file can be made unambiguous by doubling it in the compressed file.

dithering - a method of simulating a capability which does not exist.

For instance, if a graphics system has the capability to produce only three colors (red, green, or blue), then magenta may be simulated by alternating pixels of red and blue in a pattern; likewise, various shades of grey may be simulated by different patterns of black and white. Although a loss in resolution occurs with dithering, this may be insignificant compared to a gain in the virtual number of colors. [May88]

entropy - a measure of the information in a message.

"Information theory measures the amount of information in a message by the average number of bits need to encode all possible messages in an optimal encoding. ... The amount of information in a message is formally measured by the entropy of the message. The entropy is a function of the probability distribution over the set of all possible messages." [Den83: p.17]

I/O - input and output to be processed by a computer.

lossless compression - the encoding of a graphics file such that the re-created image produces a file identical to the original.

lossy compression - the encoding of a graphics file such that the re-created file produces an incomplete image of the original.

minimum run length - the minimum number of consecutive values that must be in a run for run-length encoding to be beneficial.

nibble - one-half a byte, or four bits.

pixel - one picture element in a bitmap.
"Smallest element of a display surface that can be independently referenced." [DRI85]

progressive image transmission - technique by which an image is transmitted multiple times, with each transmission consisting of a compressed image.
The earlier transmissions are highly compressed and may not be recognizable, but successive transmissions contain more definition (i.e., resolution). The advantage of such a technique is that the image may be recognizable long before transmission of higher resolutions and hence the transmission process may be halted, with an overall savings of bits transmitted and time.

redundancy - a measure of the amount of duplicated information in a file.
Redundancy is expressed as (Entropy - Average Code Length).

run - a series of consecutive values of information in a file.

run length - the number of times a value is repeated.

run-length encoding - compression technique where consecutive, identical values are replaced by the run value and the run length.
"A binary image may be represented by the set of white or black runs. This representation method is known as 'run-length coding' [and can be implemented with real-time hardware for raster scanned images.]" [Seo88]

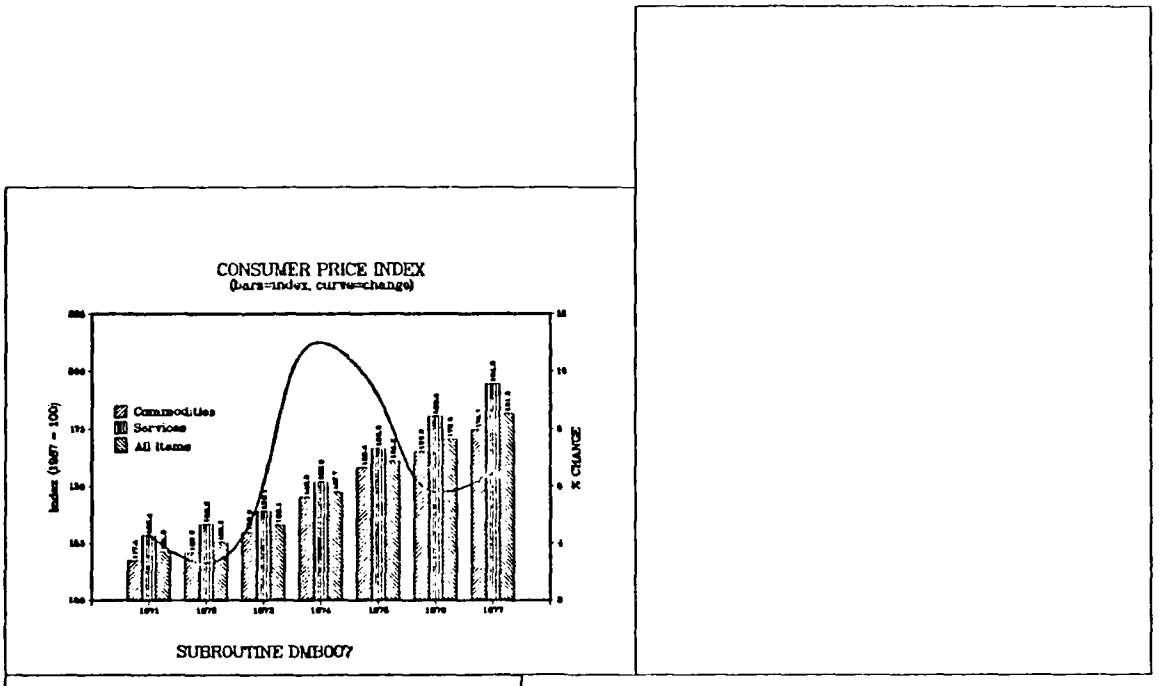
run value - that which is repeated in run-length encoding; it may be a bit, pixel of any size, a character, or pattern.

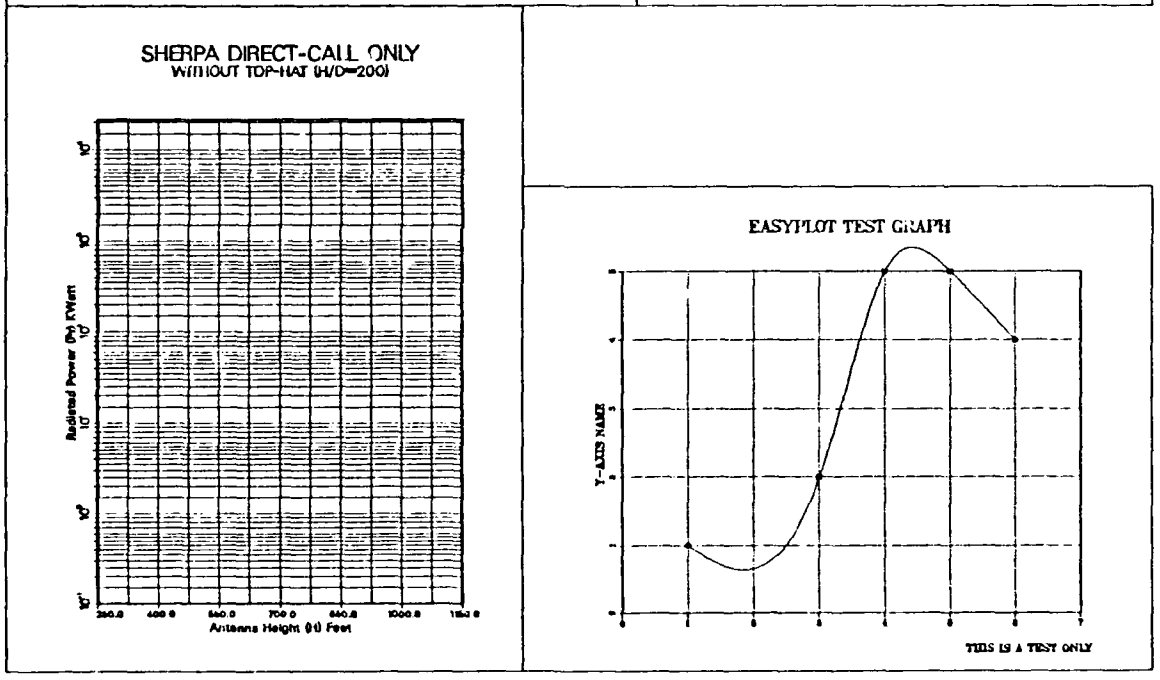
virtual screen - "Block of memory that can be addressed as if it were a memory-mapped display." [DRI85]

APPENDIX B

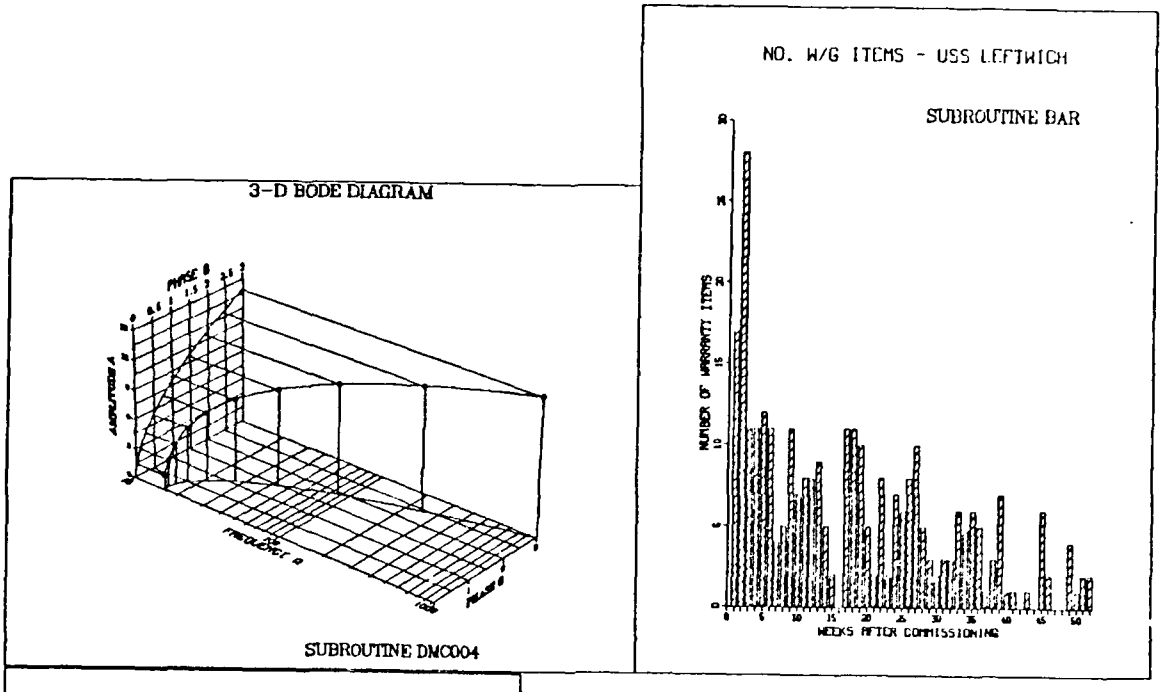
SAMPLE SET OF GRAPHS

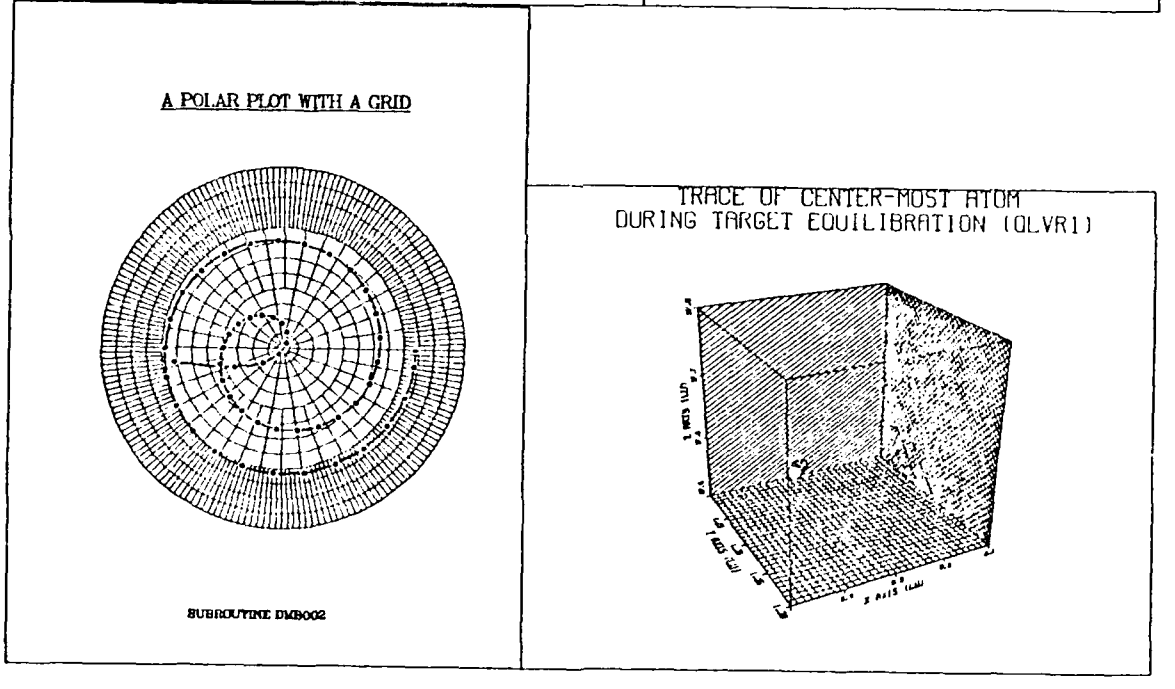
This appendix contains the set of graphs which were compressed, first by run-length encoding, then by Huffman codes, in Chapter VII. The 36 graphs are presented here in reduced format for the purpose of giving the reader an idea of the types of graphs used. They are roughly ordered by amount of compression. Graphs which were compressed more successfully are shown first.



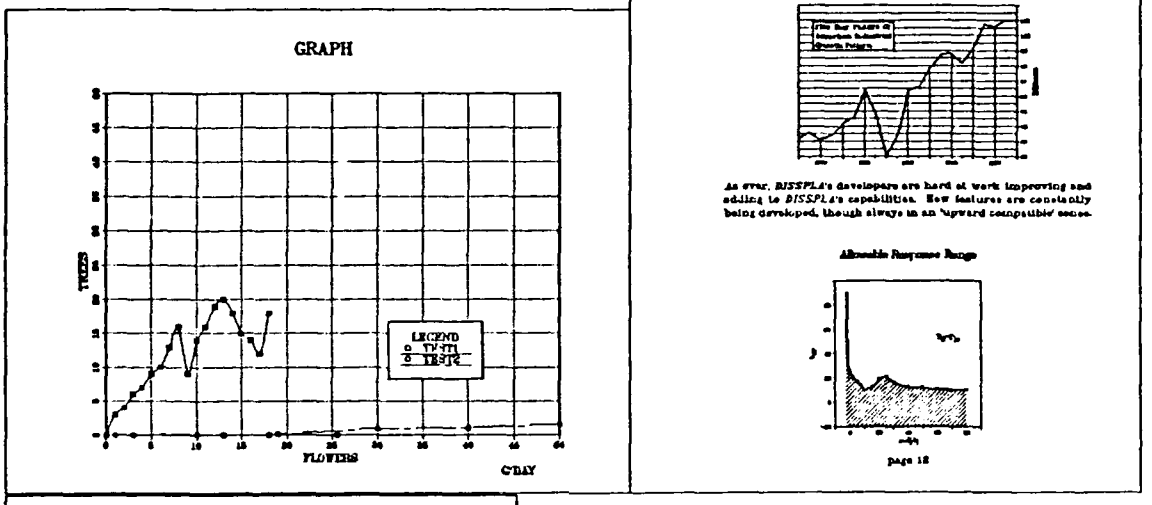


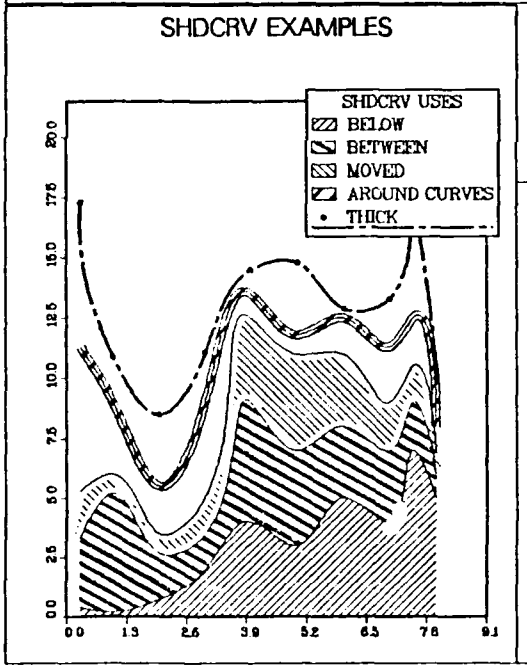
GRAPH NAME	ORIGINATOR
CPI	Computer Associates
EFRAME	J. Kretzmann
GRIDLOG	(unknown)
TESTPLOT	J. Kretzmann

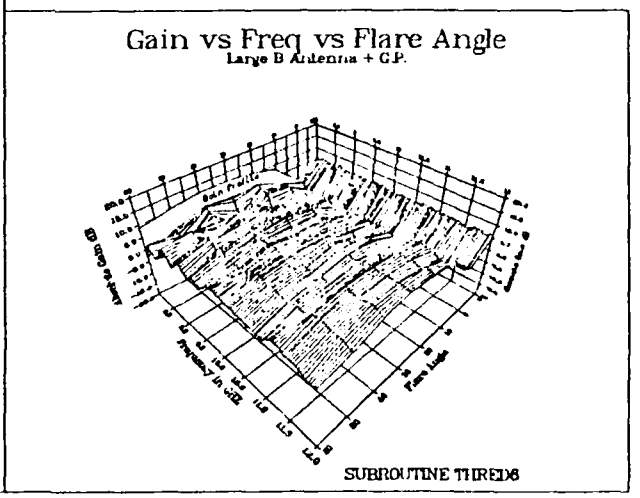




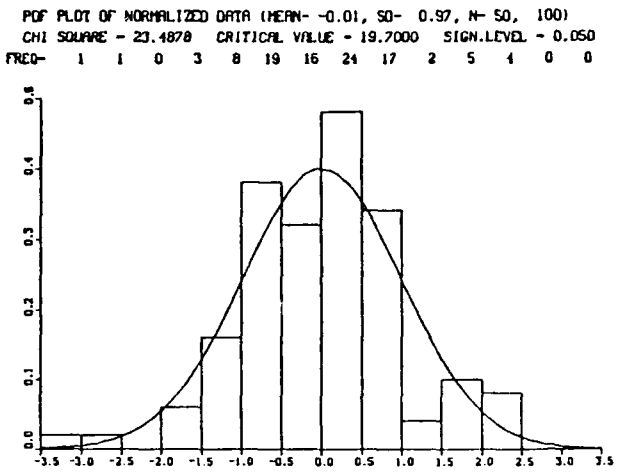
GRAPH NAME	ORIGINATOR
T3D	Computer Associates
BAR	Computer Associates
TARGT	Computer Associates
ATOMBOX	R. Rodriquez

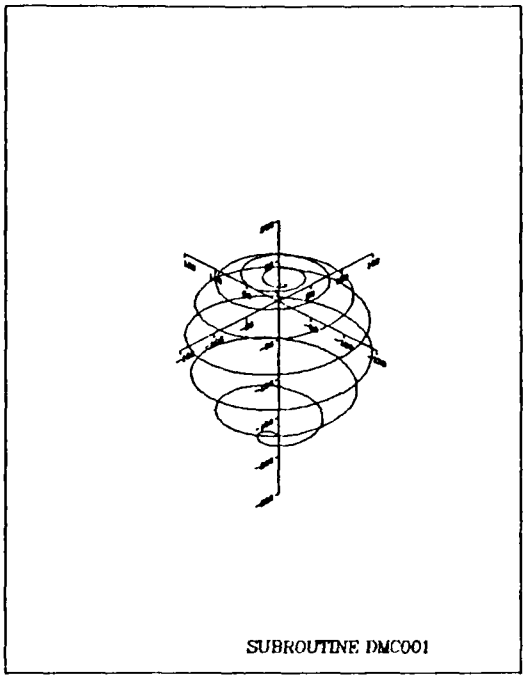


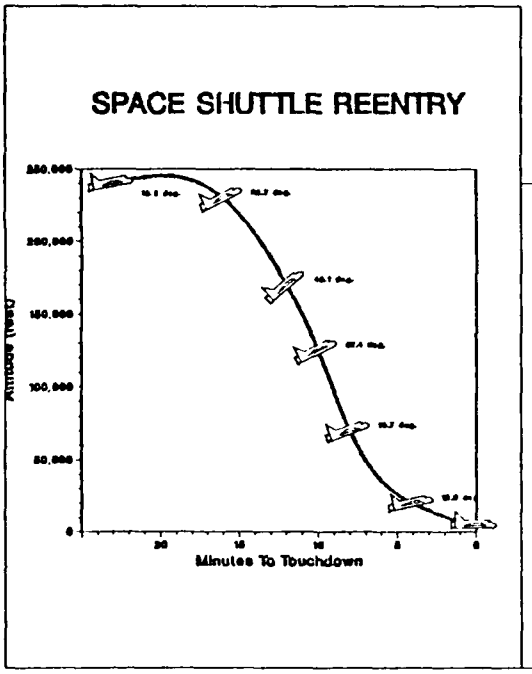


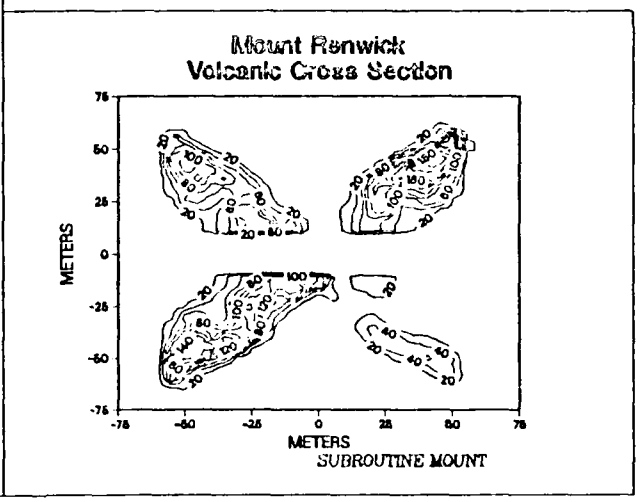


GRAPH NAME	ORIGINATOR
GDAY	J. Kretzmann
PLOTEXT	Computer Associates
SHDCRV	Computer Associates
THREED	Computer Associates

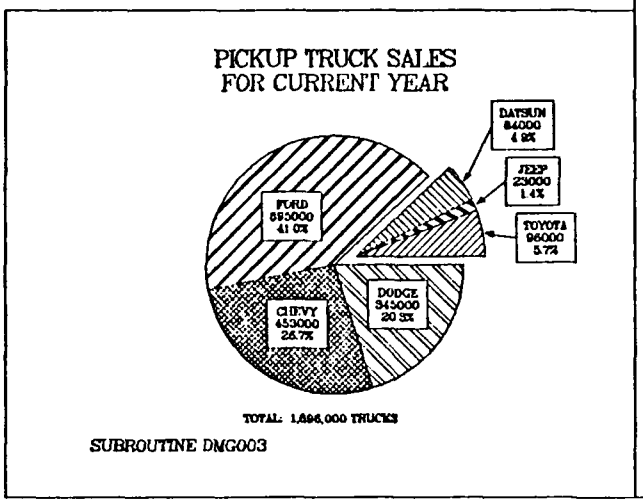


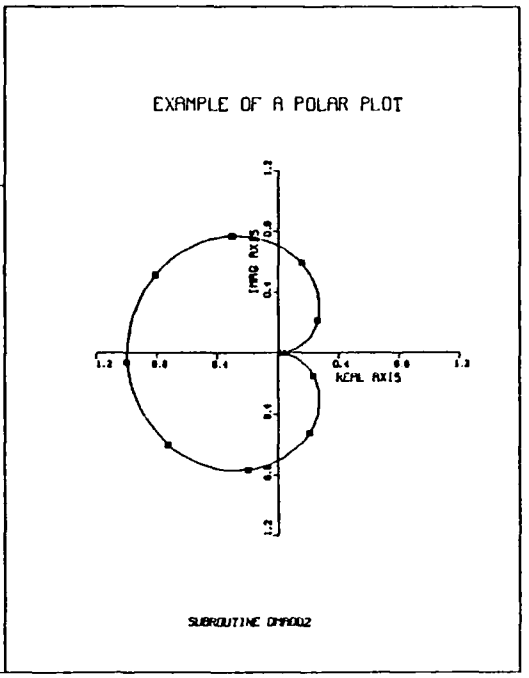


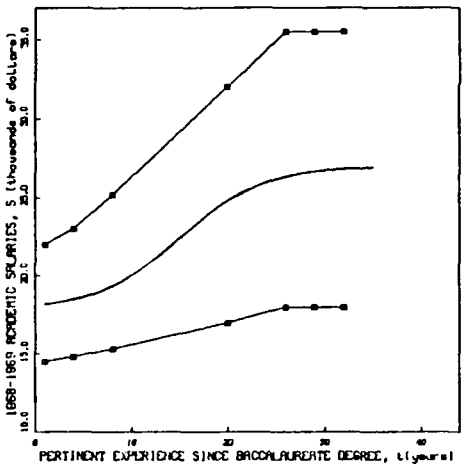


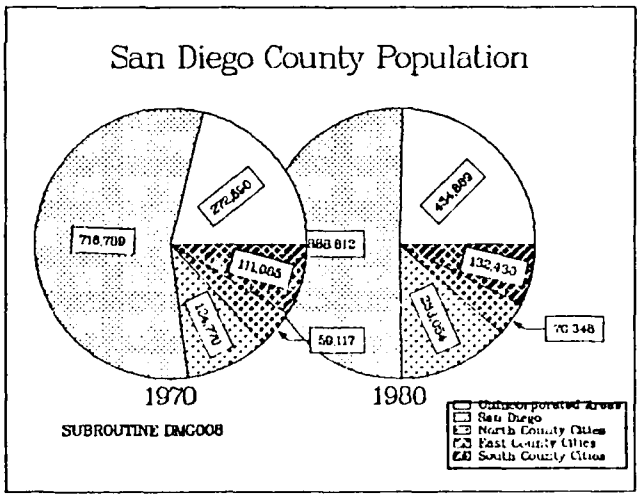


GRAPH NAME	ORIGINATOR
STEFFIN	O. Steffin
SPIRAL	Computer Associates
SHUTTLE	Computer Associates
MOUNT	Computer Associates

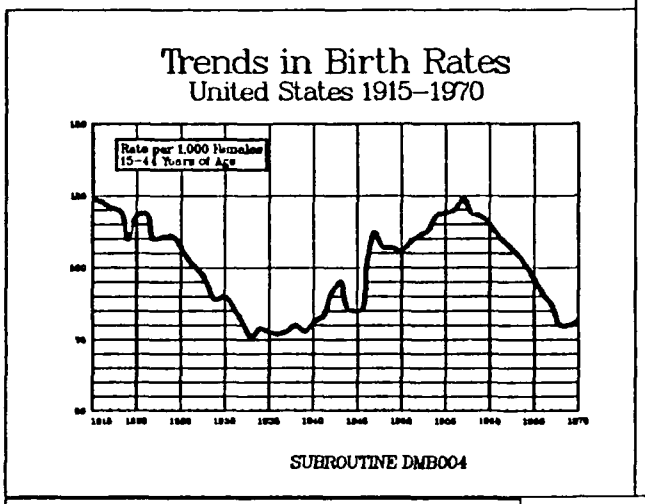


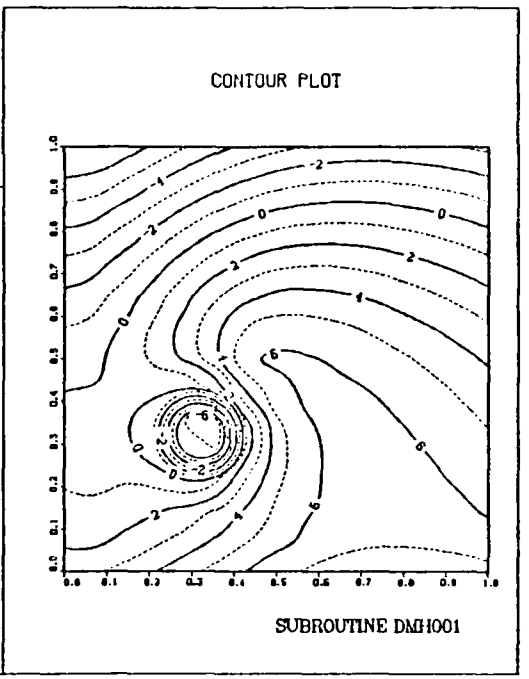


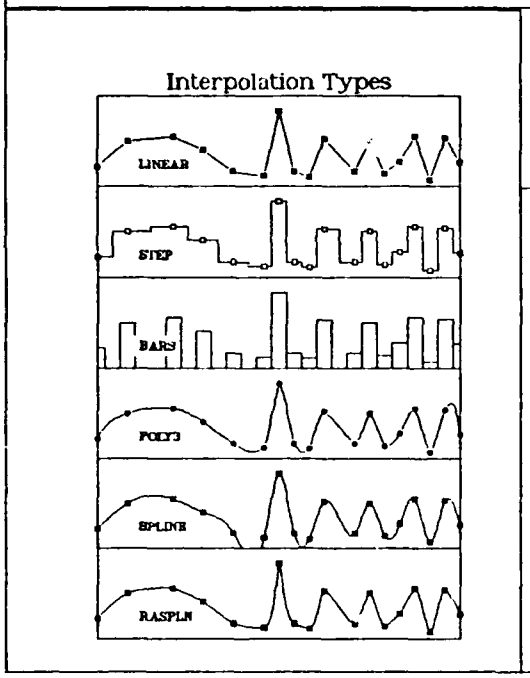


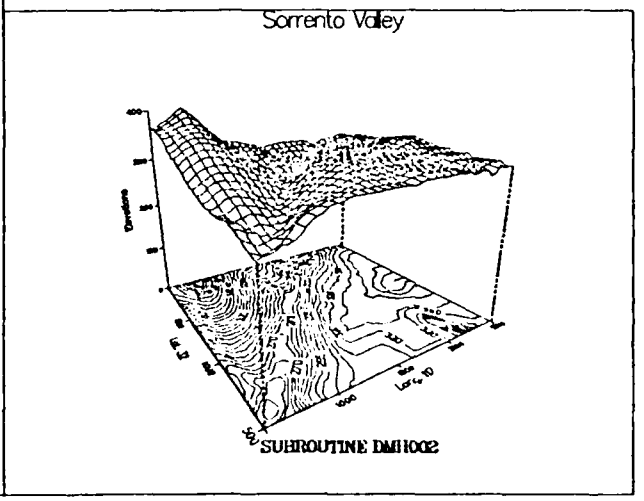


GRAPH NAME	ORIGINATOR
PIE	Computer Associates
HRT	Computer Associates
K2	R. Koehler
2PIES	Computer Associates

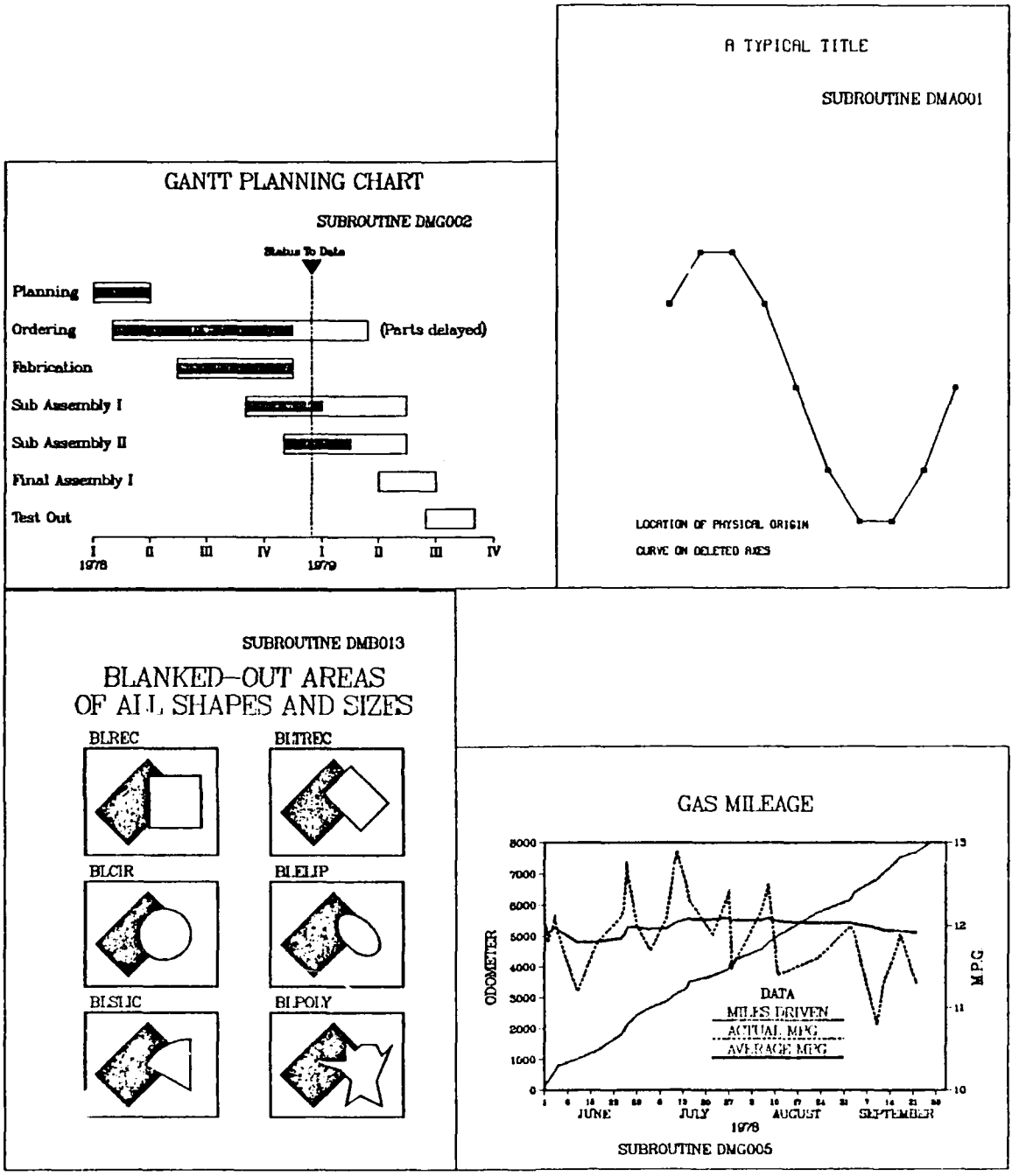




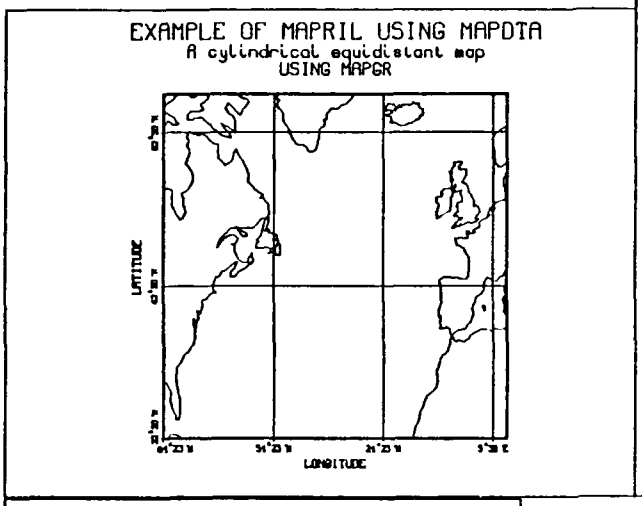


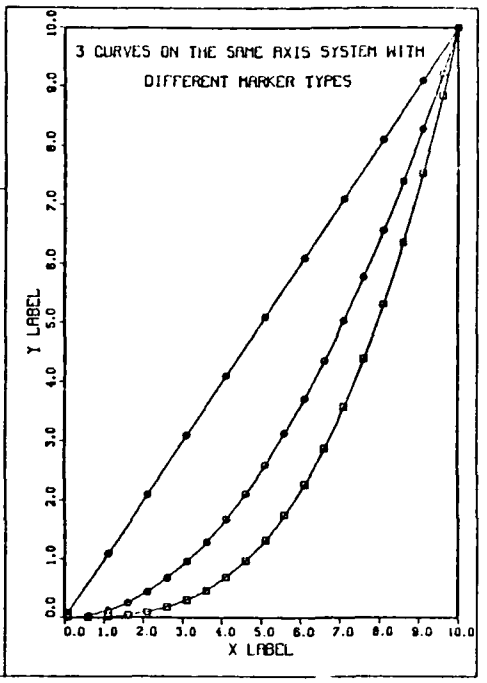


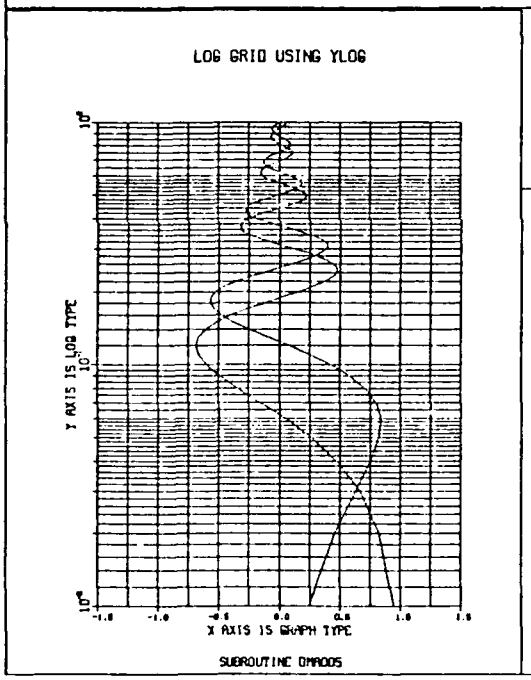
GRAPH NAME	ORIGINATOR
GRIDS	Computer Associates
CONTOUR	Computer Associates
INTERP	Computer Associates
SORRENTO	Computer Associates

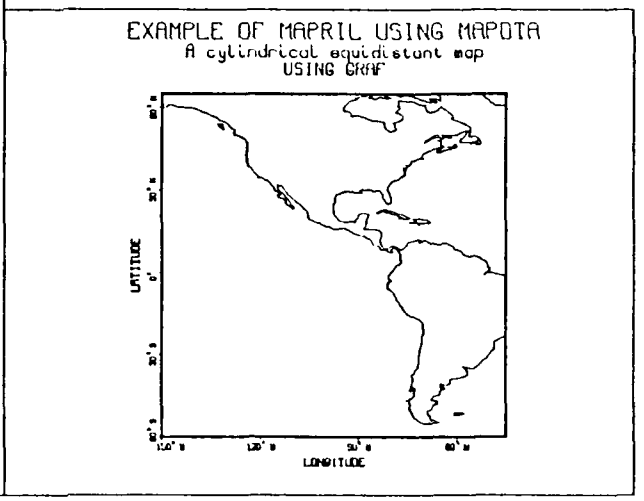


GRAPH NAME	ORIGINATOR
GANTT	Computer Associates
SIMPCRV	Computer Associates
SHAPES	Computer Associates
GAS	Computer Associates

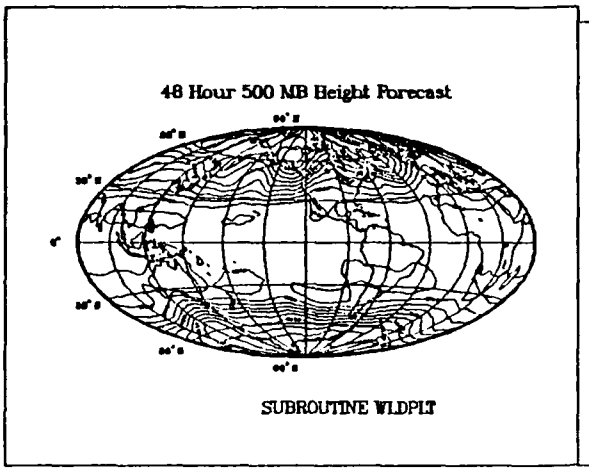


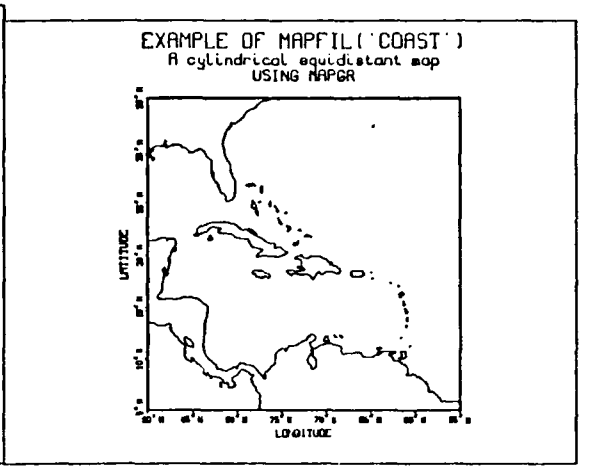


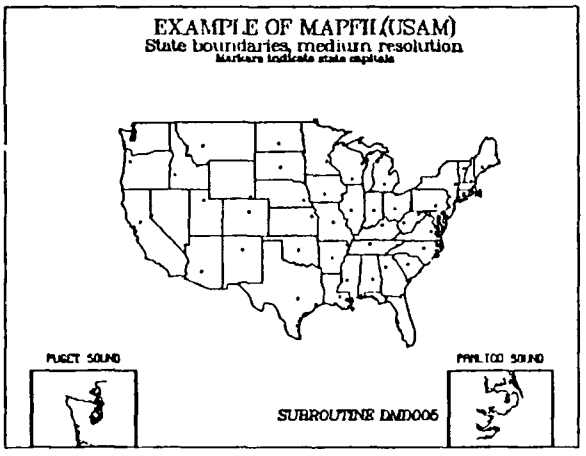


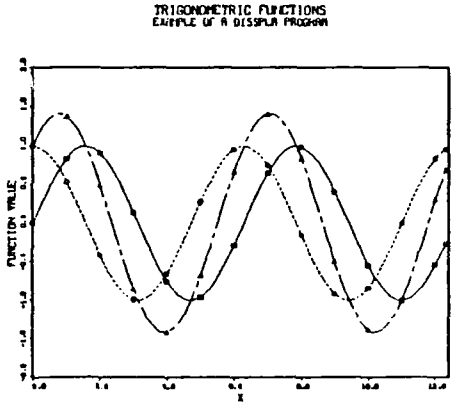


GRAPH NAME	ORIGINATOR
MAPGRID	E. Wright
3CURVES	Computer Associates
LOGCRV	Computer Associates
MAPAMER	J. Kretzmann









GRAPH NAME	ORIGINATOR
WLDPLT	Computer Associates
DREAM	J. Kretzmann
USAMAP	Computer Associates
FUNPLOT	A. Bird

APPENDIX C

COMPUTER PROGRAM LISTINGS

Three program listings follow. The first, HC EXEC, is the control program written in IBM language EXEC2. It is this program which directs the user, interactively, to input the desired set of graphs to be analyzed for compression results. It is this program which controls the "HC system."

The main program, HC FORTRAN, takes, as input, a run-length encoding (RLE) of a graph from the sample set, computes the Huffman codes for that file, and analyzes the results in terms of amount of compression. Output from each execution of HC FORTRAN is a file or listing of the Huffman coding results. Output from the system is a summary chart containing one line of results from each run of HC FORTRAN, i.e., one line for each graph in the sample set.

The third program, JSORT FORTRAN, is a subroutine which is called by HC FORTRAN to perform the sorting of each successive list of symbols. The importance of this process is evident in the explanation of derivation of Huffman codes in Chapter III. The program was adapted from two sorting routines in the IMSL Subroutine Library.

HC EXEC

```
&TRACE OFF
&IF .&1 = .D  FILEDEF 06 DISK &2 STATS E
&IF .&1 = .P  FILEDEF 06 PRINTER
&IF .&1 = .T  FILEDEF 06 TERMINAL
&IF .&1 = .   FILEDEF 06 TERMINAL

     -LOOP
     CLRSCRN
     &BEGPRINT  -END1

        PLEASE TYPE THE NAME OF THE GRAPH TO BE PROCESSED:
                      (999 to exit)
     -END1

&READ VARS &ANS
&IF .&ANS EQ .999 &GOTO -PAU
FILEDEF 02 DISK &ANS RLE E  (PERM
FILEDEF 07 DISK FILE CHART E (PERM DISP MOD
```

```
QUERY FILEDEF

EXEC RUN HC
CP SLEEP 30 SEC
&GOTO -LOOP

-PAU
&EXIT
```

HC FORTRAN

```
C  THIS PROGRAM PERFORMS THE FOLLOWING STEPS:
C
C     1- COMPUTES THE FREQUENCY OF DISTRIBUTION OF AN RLE FILE
C        (OBTAINED FROM AN 'IBMPC' RUN OF DISSPOP PROGRAM)
C
C     2- COMPUTES THE HUFFMAN CODES FOR THE SYMBOLS IN RLE FILE
C
C     3- OPTIONALLY WRITES OUTPUT FILE OR PRINTOUT OF HUFFMAN CODES
C
C     4- WRITES ENTRY IN SUMMARY OUTPUT CHART REPORT
C
C
C  Variables for this program are described below.  * indicates the
C  variable names an array.
C
C    *CHAR     - Array of TABLE characters in each transmitted record
C     ENT      - The measure of ENTROPY for the graph
C     HCAVG    - The average length of an HC for the graph
C    *HC       - The HUFFMAN CODEs for this graph
C     IPC      - Integer values of PC array
C     ITOTPC   - Integer value of TOTPC
C    *INDEX    - Pointers for Huffman coding
C     K1,2     - Pointers into Key array
C    *KINDEX   - Pointers for Huffman coding
C    *KA       - Matrix of key values used for HC computation
C    *KEY      - The sorted of indexes into sorted PC array,
C    *KOUNT    - The number of times each TABLE character is used
C     LREC     - The length of an input record (RLE)
C     NCHAR    - The total number of TABLE characters transmitted
C     NSYM     - The tolal number of symbols in a graph
C    *PC       - The calculated percent: what % is this particular
C                TABLE character of the whole?
C    *PCSAV    - Original PC array after first sorting
C     PTNAME   - The name of the plot / graph being analyzed
C    *TABLE    - Lookup table of characters whose values represent
C                the run lengths of '0' or '1' in the bitmap
C     TOTPC    - Total of the percentages; should equal 1.00
C
      CHARACTER*1  HC(20,92),CHAR(80),TABLE(92),PTNAME(10),ANS(1)
      INTEGER*4    KEY(92),KOUNT(92),NCHAR,IPC,ITOTPC,LREC
      REAL*4       PC(92),PCSAV(92),TOTPC,HCAVG,ENT
      INTEGER*2    KA(92,92),KINDEX(92),INDEX(92)
C
      DATA         PC,TOTPC,HCAVG,ENT /95*0.0/
      DATA         KA                  /8464*0/
      DATA         KINDEX             /92*1/
      DATA         INDEX              /92*20/
      DATA         HC                 /1840*' '/
      DATA         KOUNT,NCHAR,ITOTPC      /94*0/
```

```
C
      DATA           TABLE  '!','"','#','$','%','&',Z7D,'(',')','*',
     1               '+',',','-','.','/','0','1','2','3','4','5',
     2               '6','7','8','9',':',';','<','=','>','?','@',
     3               'A','B','C','D','E','F','G','H','I','J','K',
     4               'L','M','N','O','P','Q','R','S','T','U','V',
     5               'W','X','Y','Z','\',' ','^',Z81,Z82,Z83,Z84,
     6               Z85,Z86,Z87,Z88,Z89,Z91,Z92,Z93,Z94,Z95,
     7               Z96,Z97,Z98,Z99,ZA2,ZA3,ZA4,ZA5,ZA6,ZA7,
     8               ZA8,ZA9,'{','|','}','~',' '/
C
C****************  OPEN OUTPUT FILE FOR CHART REPORT  ************
C
C     OPEN (7,STATUS='NEW',FILE='CHART')
C
C****************  READ RLE FILE, RECORD BY RECORD  **************
C
      LREC = 78
      READ (2,200) (PTNAME(N),N=1,10)
                WRITE (6,*) PTNAME
      IF (PTNAME(1).EQ.'*') GOTO 130
C.........................130: write heading for "chart report"
         DO 40 I=1,500
         READ (2,200,END=50) (CHAR(N),N=1,LREC)
 200     FORMAT (80A)
C
C****************  CONSIDER EACH CHAR FROM RECORD  ***************
C
            DO 30 J=1,LREC
            NCHAR = NCHAR+1
C
C****************  SEARCH FOR CHAR IN LOOKUP TABLE  **************
C
               DO 20 K=1,92
               IF (CHAR(J).EQ.TABLE(K)) THEN
                   KOUNT(K) = KOUNT(K)+1
                   IF (K.EQ.91) GOTO 50
C..............................50: found EOP so stop freq distr
                   GOTO 30
C.............................30: found match so go for next char
               ENDIF
  20           CONTINUE
C
            WRITE (6,201) 'NO MATCH FOUND FOR CHARACTER',J,CHAR(J)
 201        FORMAT (1X,A28,I2,A1)
            GOTO 50
  30        CONTINUE
  40     CONTINUE
C
C***************  END OF STATISTICAL TABULATION  ****************
C***************  NEXT COMPUTE PERCENTS AND WRITE REPORT  *******
C

  50  WRITE (6,202) 'STATISTICS FOR PLOT: ',(PTNAME(N),N=1,10)
 202  FORMAT (//,20X,A,10A1,//)
      WRITE (6,203) 'RUN LENGTH','TABLE','COUNT','PERCENT'
 203  FORMAT (10X,A10,10X,A5,10X,A5,10X,A7)
C
      NSYM = 0
         DO 60 I=1,92
```

```
         IF (KOUNT(I).GT.0) THEN
            NSYM = NSYM + 1
            PC(NSYM) = REAL(KOUNT(I))/REAL(NCHAR)
            TOTPC = TOTPC+PC(NSYM)
            IPC   = PC(NSYM)*100.+.5001
            ITOTPC  = ITOTPC+IPC
            WRITE (6,204) NSYM,I,TABLE(I),KOUNT(I),PC(NSYM),'  =',IPC
 204        FORMAT (3X,I2,10X,I2,16X,A1,10X,I5,10X,F6.4,A,I3)
         ENDIF
  60     CONTINUE
C
      WRITE (6,205) '------','------','----'
 205  FORMAT (43X,A6,10X,A6,3X,A4)
      WRITE (6,206) NCHAR,TOTPC,ITOTPC
 206  FORMAT (43X,I6,10X,F6.4,3X,I4)
C
C*************************************************************
C  THIS SECTION REPEATEDLY SORTS LISTS OF PROBABILITIES
C  SO THAT THE HUFFMAN CODES MAY BE DERIVED
C  (THE FOLLOWING STEPS ARE REPEATED:)
C     1) SORTS THE PERCENTAGES (ASCENDING)
C     2) COMPUTES THE HUFFMAN CODES FOR EACH SYMBOL
C
C
C*************** SORT 'PC' ARRAY ASCENDING ORDER **************
C
      CALL JSORT (PC,1,NSYM,KEY)
C
C************ SAVE INITIAL SORTED % FOR REPORT *****************
C************   SET KEY AND KEY-ARRAY VALUES   *****************
C
         DO 70 I = 1,NSYM
         PCSAV(I) = PC(I)
         KEY(I)   = I
         KA(I,1)  = I
  70     CONTINUE
C
C*************** MAJOR LOOP ON REPEATED SORTED LISTS **********
C
         DO 110 J = 1,NSYM-1
C
         K1 = KEY(J)
         K2 = KEY(J+1)
C
C*************** PUT "0"|"1" INTO APPROPRIATE HC ***************
C
            DO 80 I = 1,KINDEX(K1)
            K = KA(K1,I)
            HC(INDEX(K),K) = '1'
            INDEX(K) = INDEX(K)-1
  80        CONTINUE
C
            DO 90 I = 1,KINDEX(K2)
            K = KA(K2,I)
            HC(INDEX(K),K) = '0'
            INDEX(K) = INDEX(K)-1
  90        CONTINUE
C
C************* ADD 2 LEAST ITEMS ON PC LIST *******************
C
```

```
         PC(J+1) = PC(J) + PC(J+1)
         PC(J)   = 0.0
C
C************** APPEND KEYARRAYS TO SHOW CHAINING ***************
C
            DO 100 I = 1,KINDEX(K1)
            KINDEX(K2) = KINDEX(K2)+1
            KA(K2,KINDEX(K2)) = KA(K1,I)
  100       CONTINUE
C
C************** SORT PARTIAL LIST OF PERCENTAGES ****************
C
         CALL JSORT (PC,J+1,NSYM,KEY)
C
  110    CONTINUE
C
C********** (OPTIONALLY) WRITE REPORT OF HUFFMAN CODES **********
C
CR     WRITE(6,290) 'PROBABILITY','HUFFMAN CODE'
CR290 FORMAT (1X,A,3X,A)
         DO 120 I = 1,NSYM
         LEN = 20-INDEX(I)
         HCAVG = HCAVG + PCSAV(I) * LEN
         ENT  = ENT  + PCSAV(I) * (ALOG(PCSAV(I))/ALOG(2.))
CR       WRITE(6,291) PCSAV(I),LEN,(HC(K,I),K=1,20)
CR291    FORMAT (4X,F6.4,2X,I3,20A1)
  120    CONTINUE
C
C*************** CALCULATE AVG CHAR LENGTH AND ENTROPY ***************
C
      WRITE (6,208) 'THE AVERAGE CHARACTER LENGTH (HC) IS ',HCAVG
 208  FORMAT (///,10X,A,F6.2)
      ENT = -ENT
      WRITE (6,209) 'THE ENTROPY FOR THIS PLOT IS ',ENT
 209  FORMAT (10X,A,F6.2)
      REDUN= HCAVG-ENT
      WRITE (6,209) 'THE REDUNDANCY IS ',REDUN
      COMP = (1.0-(HCAVG/8.0))*100.
      WRITE (6,209) 'THE % COMPRESSION IS ',COMP
      FACT = 8.0/HCAVG
      WRITE (6,209) 'THE COMPRESSION FACTOR IS ',FACT
      WRITE (6,210) 'THERE ARE ',NSYM,' SYMBOLS USED IN THIS PLOT.'
 210  FORMAT (/,10X,A,I2,A,/)
      GOTO 140
C
C*************** WRITE CUMULATIVE "CHART REPORT" ***************
C
 130  WRITE (7,212) 'SIZE  SYMBOLS COMPACT FACTR  HCAVG=    ENT=  REDUN'
 212  FORMAT(//,14X,A,/)
      GOTO 150
C
 140  WRITE (7,213) (PTNAME(I),I=1,10),NCHAR,NSYM,COMP,FACT,HCAVG,ENT,
     1       REDUN
 213  FORMAT(1X,10A1,3X,I6,4X,I3,2X,F5.2,3X,F3.1,3X,F4.2,6X,F4.2,3X,
     1       F4.2)
C
 150  STOP
      END
```

JSORT FORTRAN

```
C
C     SUBROUTINE JSORT
C
C     This subroutine takes as input an array(A), beginning(II)
C     and ending(JJ) subscripts which indicate that portion of the
C     array to be sorted, and a separate array(KEY) which contains
C     indices of the original array in sorted order.
C
C     PURPOSE
CCC       ADAPTED FROM NONIMSL LIBRARY ROUTINES SHSORT AND PXSORT:
CCC       KEY ADDED FROM SHSORT TO PXSORT.    ...JFK 2/89
C
C          SUBROUTINE SHSORT IS A SHELL SORT.
C          SUBROUTINE PXSORT IS INTENDED TO REARRANGE AN ARRAY OF REAL*4
C     DATA INTO ASCENDING ORDER BETWEEN TWO SPECIFIED INDICES.
C
C     ------------------------------------------------------------------
C
      SUBROUTINE JSORT(A,II,JJ,KEY)
C
      DIMENSION A(JJ),IU(16),IL(16),KEY(JJ)
      M=1
      I=II
      J=JJ
    5 IF(I .GE. J)GO TO 70
   10 K=I
      IJ=(I+J)/2
      T=A(IJ)
        IT=KEY(IJ)
      IF(A(I) .LE. T) GO TO 20
      A(IJ)=A(I)
      A(I)=T
      T=A(IJ)
        KEY(IJ)=KEY(I)
        KEY(I)=IT
        IT=KEY(IJ)
   20 L=J
      IF(A(J) .GE. T) GO TO 40
      A(IJ)=A(J)
      A(J)=T
      T=A(IJ)
        KEY(IJ)=KEY(J)
        KEY(J)=IT
        IT=KEY(IJ)
      IF(A(I) .LE. T) GO TO 40
      A(IJ)=A(I)
      A(I)=T
      T=A(IJ)
        KEY(IJ)=KEY(I)
        KEY(I)=IT
        IT=KEY(IJ)
      GO TO 40
   30 TT = A(L)
      A(L) = A(K)
      A(K)=TT
        ITT = KEY(L)
```

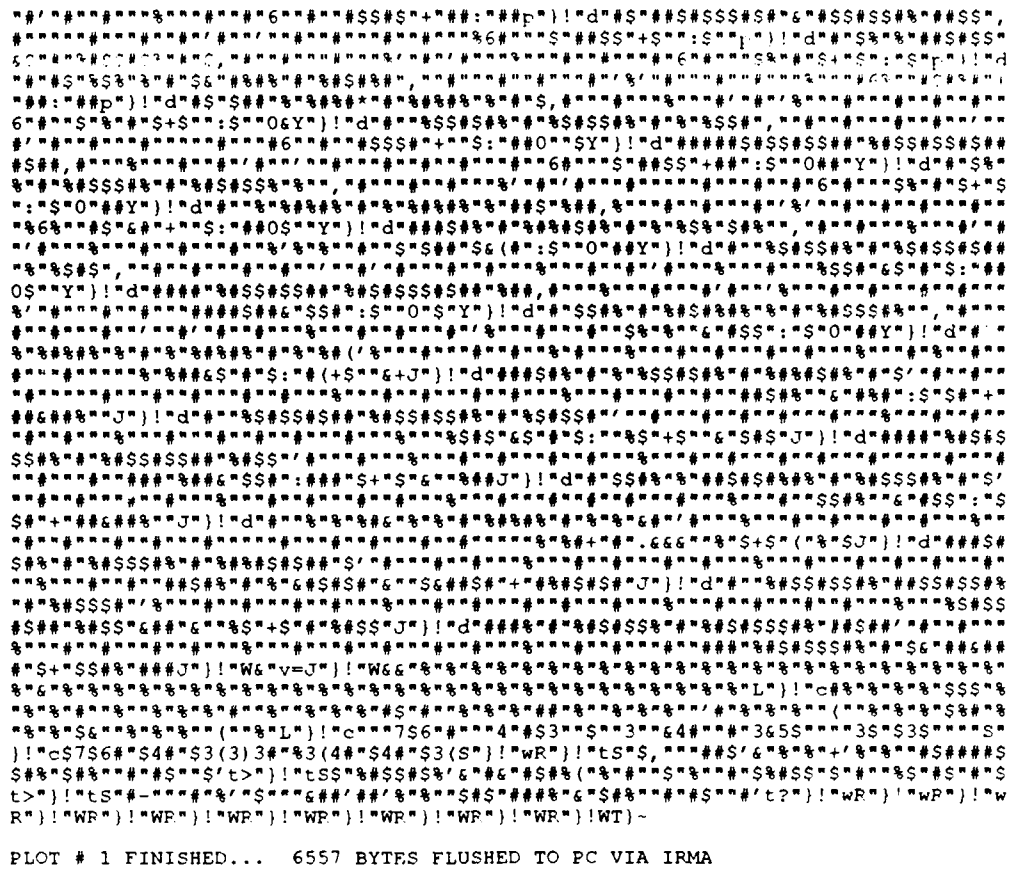
```
      KEY(L) = KEY(K)
      KEY(K)=ITT
 40 L=L-1
    IF(A(L) .GT. T) GO TO 40
 50 K=K+1
    IF(A(K) .LT. T) GO TO 50
    IF(K .LE. L) GO TO 30
    IF(L-I .LE. J-K) GO TO 60
    IL(M)=I
    IU(M)=L
    I=K
    M=M+1
    GO TO 80
 60 IL(M)=K
    IU(M)=J
    J=L
    M=M+1
    GO TO 80
 70 M=M-1
    IF(M .EQ. 0) RETURN
    I=IL(M)
    J=IU(M)
 80 IF(J-I .GE. 11)GO TO 10
    IF(I .EQ. II) GO TO 5
    I=I-1
 90 I=I+1
    IF(I .EQ. J) GO TO 70
    IF(A(I) .LE. A(I+1)) GO TO 90
    T = A(I+1)
      IT=KEY(I+1)
    K=I
100 A(K+1)=A(K)
      KEY(K+1)=KEY(K)
    K=K-1
    IF(T .LT. A(K)) GO TO 100
    A(K+1)=T
      KEY(K+1)=IT
    GO TO 90
    END
```

APPENDIX D

HC SYSTEM I/O

Included in this appendix are examples of the input and the output from the HC system described in Appendix C. The first file, BAR RLE, shows the run-length encoded file for the sample graph, BAR. The next file, BAR HC, shows the compression statistics and the Huffman codes created by HC FORTRAN. And the last file, CHART EGA, is the summary chart report showing the compression statistics for all of the sample graphs at an EGA resolution.

RLE FILE FOR 'BAR'



```
PLOT # 1 FINISHED...  6557 BYTES FLUSHED TO PC VIA IRMA
END OF DISSPOF 3.5 -- 2996 VECTORS IN 1 PLOTS.
RUN ON 1/19/89 USING SERIAL NUMBER 999 AT NAVAL POST GRADUATE
SCHOOL
```

HUFFMAN CODES AND COMPRESSION STATISTICS FOR 'BAR'

STATISTICS FOR PLOT: BAR

RUN LENGTH	TABLE	COUNT	PERCENT	LENGTH	HUFFMAN CODE
1	!	188	0.0292	12	011101100111
2	"	2652	0.4114	12	011101100110
3	#	1240	0.1924	13	0110000100000
4	$	545	0.0845	13	0110000100010
5	%	536	0.0832	13	0100010001000
6	&	109	0.0169	13	0100010001001
7	'	100	0.0155	13	0100010001111
8	(	123	0.0191	13	0100010001110
9	)	7	0.0011	13	0110000100011
10	*	6	0.0009	13	0110000100001
11	+	57	0.0088	12	010001000101
12	,	36	0.0056	12	010001001001
13	-	5	0.0008	12	010001001000
14	.	1	0.0002	12	010001000110
15	/	33	0.0051	11	01110110010
16	0	18	0.0028	11	01100001001
17	1	23	0.0036	11	01000100101
18	2	5	0.0008	10	0111010000
19	3	10	0.0016	10	0111011000
20	4	6	0.0009	10	0111010001
21	5	1	0.0002	10	0110001101
22	6	12	0.0019	10	0110000101
23	7	2	0.0003	10	0110001100
26	:	26	0.0040	10	0100010011
27	;	9	0.0014	9	011101101
28	<	1	0.0002	10	0100010000
29	=	1	0.0002	9	011101001
30	>	2	0.0003	9	011000111
31	?	1	0.0002	9	011000011
37	E	18	0.0028	8	01111011
38	F	4	0.0006	8	01111010
42	J	10	0.0016	8	01110111
44	L	42	0.0065	8	01110101
45	M	3	0.0005	8	01101111
47	O	1	0.0002	8	01101110
48	P	1	0.0002	8	01100010
50	R	34	0.0053	8	01100000
51	S	5	0.0008	8	01000101
52	T	2	0.0003	7	0111111
53	U	78	0.0121	7	0111110
55	W	20	0.0031	7	0111100
57	Y	17	0.0026	7	0110110
58	Z	2	0.0003	7	0100011
59	\	1	0.0002	6	011100
64	C	3	0.0005	6	011010
65	D	93	0.0144	6	011001
66	E	5	0.0008	6	010011
68	G	1	0.0002		010010
77	P	9	0.0014	6	010000
81	T	19	0.0029	5	01011
82	U	20	0.0031	5	01010
83	V	86	0.0133	4	0011
84	W	27	0.0042	4	0010
90	}	189	0.0293	3	000
91	~	1	0.0002	1	1

THE AVERAGE CHARACTER LENGTH (HC) IS 3.14
THE ENTROPY FOR THIS PLOT IS 3.09
THE REDUNDANCY IS 0.05
THE % COMPRESSION IS 60.80
THE COMPRESSION FACTOR IS 2.55

THERE ARE 55 SYMBOLS USED IN THIS PLOT.

CHART REPORT FROM HC SYSTEM

	SIZE	SYMBOLS	COMPACT	FACTR	LAVG=	ENT=	REDUN
3curves	7003	89	41.37	1.7	4.69	4.66	0.03
simpcrv	3843	88	42.56	1.7	4.60	4.56	0.03
interp	9061	86	42.65	1.7	4.59	4.53	0.06
shuttle	7008	89	42.74	1.7	4.58	4.55	0.03
contour	11585	88	43.75	1.8	4.50	4.46	0.04
t3d	10280	88	44.50	1.8	4.44	4.42	0.02
shapes	11197	80	44.90	1.8	4.41	4.38	0.03
funplot	6273	86	44.95	1.8	4.40	4.36	0.04
dream	7968	88	45.17	1.8	4.39	4.35	0.04
usamap	10492	88	45.76	1.8	4.34	4.30	0.03
hrt	4709	87	46.21	1.9	4.30	4.26	0.05
gas	9861	88	46.56	1.9	4.28	4.24	0.04
wldplt	12881	88	46.69	1.9	4.26	4.23	0.03
mapgrid	9783	87	46.79	1.9	4.26	4.21	0.05
spiral	5486	83	47.09	1.9	4.23	4.17	0.06
gantt	6995	86	47.64	1.9	4.19	4.17	0.02
k2	5452	87	47.94	1.9	4.16	4.12	0.04
eztest	6781	88	48.62	1.9	4.11	4.07	0.04
bar	6762	86	49.40	2.0	4.05	4.02	0.03
grids	10541	78	49.86	2.0	4.01	4.00	0.01
sorrento	12975	88	50.05	2.0	4.00	3.97	0.02
mount	12125	89	50.18	2.0	3.99	3.96	0.03
2pies	16912	86	50.70	2.0	3.94	3.90	0.04
steffin	7808	86	52.88	2.1	3.77	3.72	0.05
gday	8269	80	53.39	2.1	3.73	3.67	0.06
targt	19073	86	54.08	2.2	3.67	3.62	0.06
pie	14897	88	54.40	2.2	3.65	3.61	0.04
plotext	14948	84	54.93	2.2	3.61	3.55	0.06
shdcrv	21477	89	55.28	2.2	3.58	3.51	0.06
thred6	20995	86	58.23	2.4	3.34	3.30	0.04
atombox	25321	82	61.31	2.6	3.10	3.05	0.04
logcrv	33750	72	63.02	2.7	2.96	2.94	0.02
cpi	21022	86	63.29	2.7	2.94	2.86	0.08
gridlog	37043	78	63.36	2.7	2.93	2.91	0.02
eframe	1990	7	68.72	3.2	2.50	2.27	0.24

LIST OF REFERENCES

[Aro77] Aronson, Jules P., Data Compression -- A Comparison of Methods, National Bureau of Standards Special Publication 500-12, U.S. Government Printing Office, June 1977.

[Bac88] Bacon, Francis, "How to Quadruple Dial-Up Communications Efficiency," Mini-Micro Systems, pp. 77-81, February 1988.

[Bar88] Barnsley, Michael F. and Sloan, Alan D., "A Better Way to Compress Images," Byte, pp. 215-223, January 1988.

[Con87] Conway, Gary, Documentation for NARC.EXE, Version 1.1, Infinity Design Concepts, 1987.

[Dav88] Interview between Daniel Lee Davis, Professor, Computer Science, Naval Postgraduate School, Monterey, California, and the author, 4 February 1988.

[Dav76] Davisson, Lee D. and Gray, Robert M., Data Compression, Dowden, Hutchinson & Ross, Inc., 1976.

[Den83] Denning, Dorothy E. R., Cryptography and Data Security, Addison-Wesley, January 1983.

[DRI85] GEM (tm) Programmer's Guide, Volume 2: AES, Digital Research Inc., 1985.

[Eub89] Telephone conversation between Gordon E. Eubanks, President of Symantec, Inc., Cupertino, California, and the author, April 1989.

[Fan49] Fano, R. M., "The Transmission of Information," Technical Report No. 65, Research Laboratory of Electronics, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1949.

[Gun84] Gunning, Michael, Documentation for GRAFPC, Version 1.0, Naval Postgraduate School, Monterey, California, 1984.

[Ham86] Hamming, Richard W., Coding and Information Theory, Prentice-Hall, 1986.

[Hel87] Held, Gilbert, Data Compression, John Wiley & Sons Ltd., 1987.

[Huf52] Huffman, David A., "A Method for the Construction of Minimum-Redundancy Codes." Proceedings of the IEEE, IRE, vol. 40, no. 9, pp. 1098-1101, 1952.

[Lel88] Lelewer, Debra A. and Hirschberg, Daniel S., "Data Compression," ACM Computing Surveys, vol. 19, no. 3, pp. 261-296, 3 September, 1988.

[May88] Interview between Michael M. Mayer, Lieutenant, USN, Naval Postgraduate School, Monterey, California, and the author, 24 October 1988.

[Mur88a] Murphy, J. L., Helman, D. R., and Hitchner, L, E., "Managing Digital Images in a Network Environment," SPIE, vol. 900, pp. 14-17, 1988.

[Mur88b] Interview between James L. Murphy, Professor, Computer Engineering, University of California at Santa Cruz, and the author, 16 September 1988.

[Ped89] Peddie, Jon, "Focus: High Performance Cards for PCs," Computer Graphics Today, vol. 6, no. 2, pp. 7,8, February 1989.

[Pet87] Peterson, Ivars, "Packing It In," Science News, vol. 131, no. 18, pp. 272-288, 2 May 1987.

[Seo88] Seong-Dae Kim, Jeong-Hwan Lee, and Jae-Kyoon Kim., "A New Chain-Coding Algorithm for Binary Images Using Run-Length Codes," CVGIP, pp. 114-128, January 1988.

[SGI] GT Graphics Library User's Guide, Version 1.0, Silicon Graphics, Inc., document no. 007-1202-010

[Sha48] Shannon, C. E., "A Mathematical Theory of Communication," Bell Systems Technical Journal, vol. 27, pp.398-403, July, 1948.

[Tan81] Tanenbaum, Andrew S., Computer Networks, Prentice-Hall, Inc., 1981.

[Win88] Winiarski, Kathryn, "Fractals Library Could Change Visualization," Computer Graphics Today, vol. 5, no. 4, pp. 1,27, April 1988.

[Wil88] Wilkes, Derek, "Monitors: Meeting High Resolution Needs," Computer Graphics Today, pp.10-12, December 1988.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Uno R. Kodres, Code 52kr Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	2
4.	Douglas G. Williams, Code 0141 W.R. Church Computer Center Naval Postgraduate School Monterey, California 94943	2
5.	Lt. Michael M. Mayer SMC 2836 Naval Postgraduate School Monterey, California 93943	1
6.	Michael Gunning Hewlett-Packard 3404 East Harmony Road Ft. Collins, Colorado 80525	1
7.	Gordon E. Eubanks, Jr. SYMANTEC 10201 Torre Avenue Cupertino, California 95014-2132	1
8.	Daniel L. Davis MBARI 160 Central Avenue Pacific Grove, California 93950	1
9.	Jane L. Kretzmann 1113 Moorefield Hill Court Vienna, Virginia 22180	6